Certifying Combinatorial Optimization:
A Unified Approach Using Pseudo-Boolean Reasoning

by Andy Oertel

Thesis for the degree of Doctor of Philosophy in Engineering

Thesis advisors: Prof. Jakob Nordström, Asst. Prof. Susanna F. de Rezende
Faculty opponent: Prof. Randal E. Bryant

To be presented, with the permission of the Faculty of Engineering of Lund University, for public criticism in E:1406, Ole Römers väg 3 at the Department of Computer Science on May 29, 2026 at 13:30.

DOKUMENTDATABLAD enl SIS 61 41 21

Organization
LUND UNIVERSITY

Department of Computer Science
Box 118
SE–221 00 LUND
Sweden

Document name
DOCTORAL DISSERTATION

Date of disputation
2026-05-29

Sponsoring organization
Wallenberg AI, Autonomous Systems and Software Program (WASP) funded by the Knut and Alice Wallenberg Foundation

Author(s)
Andy Oertel

Title and subtitle
Certifying Combinatorial Optimization: A Unified Approach Using Pseudo-Boolean Reasoning

Abstract
Combinatorial optimization is a powerful way to solve complex problems, like planning, scheduling, or hardware verification, by expressing the problem in a mathematical form using discrete variables that can be solved by general solvers. Due to major advances in algorithms for solving combinatorial optimization problems, these solvers can tackle real-world challenges efficiently. However, as solvers become more powerful, they also become larger and more complex, which makes it harder to trust that their output is correct. Ensuring that the solver gives a correct answer becomes especially important when mistakes could have serious consequences, e.g., when solvers are used to match organ donors and recipients or dispatch ambulances.

Testing the solver, which verifies correctness only on known input-output pairs, provides no guarantee that the solver returns correct answers on untested inputs and therefore we can not fully trust that the answer is correct. Formal verification can prove that a solver adheres to a formal specification and thus guarantees that the answer of the solver is correct, but this approach remains infeasible for modern solvers. The approach that has proven most effective for providing correctness guarantees for solver outputs is certifying algorithms. The idea behind certifying algorithms is that the algorithm generates a certificate that shows the correctness of the result. An independent tool can then use the certificate to verify that the result is correct with respect to the input. This verification tool can be simple enough to enable formal verification of its correctness, ensuring that its verdict can be trusted.

This thesis presents the first viable certification approach for several combinatorial optimization solvers that had previously been considered out of reach. This is achieved through a multipurpose certification system built on so-called pseudo-Boolean reasoning, which enables the generation of correctness certificates across a these wide range of different solver paradigms. Developing a multipurpose system allows the checker to be reused for all types of solvers, which sets our work apart from previous, more specialized approaches. Although we use pseudo-Boolean reasoning to certify the solver output, the solver itself does not need to perform pseudo-Boolean reasoning, and making a solver certifying does not require any changes to its internal reasoning. To have also developed a checker that is formally verified to be correct to ensure that this checker can be truster.

Key words
certifying algorithms, combinatorial optimization, proof logging, MaxSAT, 0-1 integer linear programming, pseudo-Boolean optimization, automated reasoning

Classification system and/or index terms (if any)

Supplementary bibliographical information

Language
English

ISSN and key title
1404-1219

ISBN
978-91-8104-995-4 (print)
978-91-8104-994-7 (pdf)

Recipient's notes

Number of pages
347

Price

Security classification

I, the undersigned, being the copyright owner of the abstract of the above-mentioned dissertation, hereby grant to all reference sources the permission to publish and disseminate the abstract of the above-mentioned dissertation.

Signature ____________________ Date 2026-04-27

Certifying Combinatorial Optimization:
A Unified Approach Using Pseudo-Boolean Reasoning

by Andy Oertel

LUND
UNIVERSITY

A doctoral thesis at a university in Sweden takes either the form of a single, cohesive research study (monograph) or a summary of research papers (compilation thesis), which the doctoral student has written alone or together with one or several other author(s).
In the latter case the thesis consists of two parts. An introductory text puts the research work into context and summarizes the main points of the papers. Then, the research publications themselves are reproduced, together with a description of the individual contributions of the authors. The research papers may either have been already published or are manuscripts at various stages (in press, submitted, or in draft).

Funding information: Andy Oertel was funded by the Wallenberg AI, Autonomous Systems and Software Program (WASP) funded by the Knut and Alice Wallenberg Foundation.

ISBN 978-91-8104-994-7 (electronic version)
ISBN 978-91-8104-995-4 (print version)
ISSN 1404-1219
Doctoral Dissertation 88, 2026
LU-CS-DISS: 2026-05

Lund University
Department of Computer Science
Box 118
SE–221 00 LUND
Sweden

E-mail: andy.oertel@cs.lth.se
WWW: aoertel.de

Typeset using LaTeX
Printed in Sweden by Tryckeriet i E-huset, Lund, 2026

© *Andy Oertel 2026*

Contents

Acknowledgements xi

Popular Science Summary xiii

Contribution Statement xv

Certifying Combinatorial Optimization: A Unified Approach 1

1 Introduction 1
2 Background 3
2.1 Basic Notation 3
2.2 Combinatorial Optimization 3
2.2.1 Boolean Satisfiability (SAT) 5
2.2.2 Maximum Satisfiability 6
2.2.3 Pseudo-Boolean Optimization 9
2.2.4 Preprocessing 10
2.2.5 Enumeration and Counting Problems 11
2.3 Proof Complexity and Proof Systems 11
2.4 Certifying Algorithms 14
2.4.1 Certifying Algorithms for SAT 16
3 Related Work 19
4 Pseudo-Boolean Certificates 21
4.1 Motivation 21
4.2 Our Pseudo-Boolean Proof System 23
4.2.1 Proof Configuration 23
4.2.2 Rules from Previous Work 26
4.2.3 Extensions to the System by This Thesis 27
4.3 Pseudo-Boolean Proof Checking Tool 38
4.3.1 Constraint Data Structures 38
4.3.2 Reverse Unit Propagation 39
4.3.3 Syntactic Implication 40
4.3.4 Implicational Subproofs 41
5 Main Results of Included Papers 41
5.1 Summary of Paper I 42
5.2 Summary of Paper II 43
5.3 Summary of Paper III 44
5.4 Summary of Paper IV 45

5.5 Summary of Paper V . . . 46
5.6 Summary of Paper VI . . . 47
5.7 Summary of Paper VII . . . 48
5.8 Summary of Paper VIII . . . 49
5.9 Further Contributions . . . 50
6 Conclusions and Future Work . . . 50
6.1 Short Term Future Work . . . 51
6.2 Long Term Future Work . . . 51
References . . . 52

Included Papers 71

I Certified CNF Translations for Pseudo-Boolean Solving 73
1 Introduction . . . 73
2 Preliminaries . . . 76
3 Certified CNF Translation Using the Sequential Counter Encoding 78
4 A General Framework for Certifying CNF Translations . . . 84
5 Certifying the Binary Adder Network Encoding . . . 92
6 Certifying the Totalizer and Generalized Totalizer Encodings . . . 94
7 Experimental Evaluation . . . 96
7.1 Benchmarks . . . 97
7.2 End-to-End Solving and Verification . . . 97
7.3 Translation and Verification . . . 100
7.4 Overhead of Proof Logging . . . 101
7.5 Comparison with PB Solvers . . . 102
7.6 Certifying MaxSAT Optimal Values . . . 103
8 Concluding Remarks . . . 107
References . . . 109

II Certified Core-Guided MaxSAT Solving 117
1 Introduction . . . 117
1.1 Previous Work . . . 118
1.2 Our Contributions . . . 119
1.3 Outline of This Paper . . . 119
2 Preliminaries . . . 120
3 The OLL Algorithm for Core-Guided MaxSAT Solving . . . 121
4 Proof Logging for the OLL Algorithm for MaxSAT . . . 123
5 Experimental Evaluation . . . 129
6 Concluding Remarks . . . 131
References . . . 133

III Certifying Without Loss of Generality Reasoning in SIS for MaxSAT 141
1 Introduction . . . 141
2 Preliminaries . . . 146
3 The Dynamic Polynomial Watchdog Encoding for SIS . . . 149
3.1 Initialization . . . 149
3.2 Coarse Convergence Phase . . . 150

3.3 Fine Convergence Phase 150
3.4 Stratification 151
4 Certifying Solution-Improving MaxSAT with the DPW Encoding . 151
4.1 Proof Logging for Clauses of the DPW Encoding 152
4.2 Proofs Without Loss of Generality Using Shadow Circuits . 152
4.3 Stratification 155
4.4 Limiting the Use of Shadow Circuits 155
4.5 Discussion of Simpler Approach and Why It Does Not Work 156
5 Experimental Evaluation 157
6 Conclusion 159
Appendix A Formalization of the Proof Logging of SIS with the DPW 160
A.1 Coarse Convergence 160
A.2 Fine Convergence 161
A.3 Conclusion of Optimality 162
Appendix B Proof Logging of Additional Techniques in Pacose 163
B.1 TrimMaxSAT 164
B.2 Hardening 165
Appendix C Additional Experimental Evaluation 165
C.1 Binary Adder Encoding and Encoding Selection Heuristic . 165
C.2 Coarse Convergence with Assumptions Instead of Clauses . 166
C.3 Proof Logging Overhead Analysis 167
References 168

IV Certifying MIP-Based Presolve Reductions for 0–1 ILP 175
1 Introduction 175
2 Pseudo-Boolean Proof Logging with VeriPB 177
2.1 Pseudo-Boolean Reasoning with the Cutting Planes Method 177
2.2 A New Rule for Objective Function Updates 178
3 Certifying Presolve Reductions 179
3.1 General Techniques 180
3.2 Primal Reductions 181
3.3 Dual Reductions 183
3.4 Example 184
4 Computational Study 185
4.1 Experimental Setup 185
4.2 Overhead of Proof Logging 186
4.3 Verification Performance on Presolve Certificates 186
4.4 Performance Analysis on Constraint Propagation 187
5 Conclusion 189
References 190

V End-to-End Verification for Subgraph Solving 195
1 Introduction 195
1.1 Our Contribution 197
1.2 Comparison to Related Work 198
1.3 Outline of This Paper 198
2 Preliminaries 198

3 Formally Verified Graph Proof Checkers 200
3.1 Verified Pseudo-Boolean Proof Checking 200
3.2 Verified Graph Problem Encoders 202
3.3 End-to-End Verification 203
4 Proof Elaboration 204
4.1 Lining up Encodings 205
4.2 Elaborating on Syntactic Sugar 206
5 Experiments 208
6 Conclusion 209
References 210

VI Certified MaxSAT Preprocessing 215
1 Introduction 215
1.1 Previous Work 216
1.2 Our Contribution 216
1.3 Organization of This Paper 217
2 Preliminaries 217
2.1 Pseudo-Boolean Proof Logging Using Cutting Planes 218
2.2 Maximum Satisfiability 219
3 Proof Logging for MaxSAT Preprocessing 220
3.1 Overview 220
3.2 Worked Example of Certified Preprocessing 224
4 Verified Proof Checking for Preprocessing Proofs 225
4.1 Output Section for Pseudo-Boolean Proofs 226
4.2 Verified Proof Checking for Reformulations 227
4.3 Verified WCNF Frontend 228
5 Experiments 229
6 Conclusion 231
Appendix A Complete Proof Logging for MaxSAT Preprocessing . . . 232
A.1 Fixing Variables 232
A.2 Preprocessing on the Initial WCNF Representation 232
A.3 Preprocessing on Objective-Centric Representation 235
A.4 Conversion to WCNF — Renaming Variables 237
A.5 On Solution Reconstruction and Instances Solved 238
References 239

VII Faster Certified Symmetry Breaking Using Orders With Auxiliary Variables 247
1 Introduction 247
2 Preliminaries 250
2.1 The VeriPB Proof System 250
2.2 Symmetry Breaking 252
3 Strengthening with Auxiliary Variables 253
3.1 Specifications 254
3.2 Orders with Auxiliary Variables 254
3.3 Validity 255
3.4 Dominance-Based Strengthening Rule 256

3.5 Redundance-Based Strengthening Rule 257
4 Efficient Proof Logging in SATSUMA 257
5 Proof Checker Implementation 258
6 Experimental Evaluation 258
7 Concluding Remarks 261
Appendix A The Cutting Planes Proof System 262
Appendix B Full Description of Extended Proof System 264
B.1 Specifications 265
B.2 Orders with Auxiliary Variables 266
B.3 Validity 267
B.4 Dominance-Based Strengthening Rule 268
B.5 Redundance-Based Strengthening Rule 270
Appendix C Proof Logging in Satsuma 271
C.1 Defining the Lexicographical Order 271
C.2 Deriving the Symmetry-Breaking Clauses 276
Appendix D Example Of Symmetry Breaking Proof 284
Appendix E Details on Crafted Benchmarks 298
References 299

VIII Proof Logging for Projected Enumeration (and Counting?) Problems 303
1 Introduction 303
2 Proof Logging Beyond Implicational Proofs of Unsatisfiability . . . 305
2.1 Strengthening 305
2.2 Deletions 306
2.3 Enumeration Proofs with Strengthening and Deletions . . . 307
3 Projected Enumeration Proofs in VERIPB 308
3.1 Formal Verification of Enumeration Proofs 312
3.2 Experiments for Enumeration Proofs 314
4 Proofs for Preprocessing and Counting Without Enumerating . . . 315
4.1 Tabulating Solutions as Extension Variables 316
4.2 Extending the Preserved Set 317
4.3 Shrinking the Preserved Set 318
4.4 Deleting Remaining Constraints 319
5 Conclusion 320
References 320

Acknowledgements

I would like to take this opportunity to thank my family for always supporting me on my educational journey. Without my parents, sister, and brother I would not have made it this far. I especially also want to thank Johanna for all the time we spent together and will spend together. Thank you for always encouraging me and your unwavering support. Martin, Max, and Sebastian, I am very happy to have you as my friends and enjoy every moment we spend together.

When it comes to the work environment, many thanks to Benjamin, Christophe, David, Duri, Gaia, Jonas, Kilian, Morgan, Noel, Rui, Shuo, Stefan, Wietze, and Yassine for your company in and outside of office. I would also like to thank the people in the CP, OR, and SAT communities for welcoming me. I am happy to be friends with many of you.

I would also like to thank the administration at the Department of Computer Science of LTH for always being very helpful. Also, thank you, Mikkel and Anne for creating such a lovely research environment in Copenhagen. I always enjoyed being at BARC and joining the events you organized.

I would like to thank all my co-authors and everyone who worked on VeriPB for helping me with this endeavour. I would not have been able to do all this work I am presenting in this thesis without you. The discussions with all of you were always fruitful and enlightening. Especially, I would like to thank Bart Bogaerts, Stefan Gocht, Ciaran McCreesh, and Jakob Nordström for designing the original VeriPB proof system, which my work builds upon.

I would also like to thank the Wallenberg AI, Autonomous Systems and Software Program (WASP) funded by the Knut and Alice Wallenberg Foundation for funding my research and my stay in Berkeley. Additionally, I would like to thank the SAT association, Zuse Institute Berlin, Landshövding Per Westlings Minnesfond, COST action *EuroProofNet*, and CADE conference for providing me with additional travel funding. For hosting the computational resources to run my experiments, I would like to thank LUNARC at Lund University.

Last but not least, I thank my PhD supervisor Jakob Nordström for introducing me to the topic, proposing all kinds of ideas, and making all of this possible. I also thank my co-supervisor Susanna F. de Rezende for supporting me during my PhD.

Thank you very much!

Tack så mycket!

Vielen Dank!

Popular Science Summary

Every day, computers help humans to make decision and sometimes even make such decisions on their own. For instance, automated systems can be used to plan work schedules so that they are as fair as possible while accommodating all constraints, or optimize the logistics of a company to be as cheap or fuel efficient as possible. However, how can we be sure that these decisions are the best or even correct, e.g., that all constraints for a work schedule are respected?

Let us consider a smaller example to illustrate the challenges and how this thesis addresses them. Imagine that you want to go camping with a friend in Sweden for seven days. Since you are canoeing and there is no supermarket in the wilderness, you have to plan what food and how much to buy in advance. You also want to save money, so you want to buy enough food for two persons lasting seven days for the cheapest possible price. For simplicity, consider only the requirement to reach the recommended daily intake of protein and carbohydrate of 50 g protein and 290 g carbohydrates per person. Hence, in total you need to buy food that contains 700 g protein and 4060 g carbohydrates. The store only has 1 kg packs of rice, peanuts, and lentils in stock, where the nutritional value and price of each pack is listed in Table 1. You can only buy whole packs.

Food	Price	Protein	Carbohydrates
Rice	60 kr	100 g	750 g
Peanuts	40 kr	300 g	150 g
Lentils	50 kr	200 g	450 g

Table 1: *Price in Swedish krona (*kr*) and nutrition for the food available in the store. Each package has a size of* 1 kg.

You might be intrigued to ask an artificial intelligence chatbot for the solution, and it might tell you that the cheapest option is to buy 4 kg and of rice and 3 kg of lentils for 390 kr.[1] But how do you know that this is the best answer for this problem? First, we should make sure that this answer actually provides enough nutrition and costs 390 kr. Using a calculator, we can check that the food has $4 \cdot 100\,\text{g} + 3 \cdot 200\,\text{g} = 1000\,\text{g}$ protein and $4 \cdot 750\,\text{g} + 3 \cdot 450\,\text{g} = 4350\,\text{g}$ carbohydrates, which is indeed sufficient. Furthermore, we can check that the price is indeed $4 \cdot 60\,\text{kr} + 3 \cdot 50\,\text{kr} = 390\,\text{kr}$. So we know that we can buy the food for 390 kr, but is this really the cheapest possible option? The chatbot will probably try to argue something complicated to justify that this is correct, but do you trust this?

Let me tell you that the actual cheapest option is to buy 5 kg rice and 1 kg lentils for 350 kr. Using a calculator, we can confirm that the food has $5 \cdot 100\,\text{g} + 200g = 700\,\text{g}$

[1] This is the actual answer of GPT-5.2 for this problem at the time of writing.

protein and $5 \cdot 750\,\text{g} + 450\,\text{g} = 4200\,\text{g}$ carbohydrates, which is sufficient for the seven days. In total, we pay $5 \cdot 60\,\text{kr} + 50\,\text{kr} = 350\,\text{kr}$. So we now know that the chatbot was incorrect in claiming that the cheapest possible options is 390 kr and that we can buy the required food for 350 kr.

We have already seen one invalid claim of optimality from the chatbot. So why should it not be possible to buy enough food for less than 350 kr? The reason we know that 350 kr is optimal is that we can prove that it is necessary to pay *at least* 350 kr to reach the nutrition goals. One way to show this would be to iterate through all possible shopping combinations below 350 kr to see that they do not provide enough nutrients, which might be doable for this example, but there are too many combinations when the problem gets more complex. Instead, the following simple mathematical argument shows that the price is at least 350 kr.

We can consider the variable r, p, and ℓ for how many 1 kg packs of rice, peanuts, and lentils we buy, respectively. To buy enough protein, it has to hold that

$$100\,\text{g} \cdot r + 300\,\text{g} \cdot p + 200\,\text{g} \cdot \ell \geq 700\,\text{g} \tag{1}$$

and to buy enough carbohydrates, it has to hold that

$$750\,\text{g} \cdot r + 150\,\text{g} \cdot p + 450\,\text{g} \cdot \ell \geq 4060\,\text{g}\,. \tag{2}$$

Dividing inequality (1) by 100 g results in

$$\begin{aligned} 100\,\text{g} \cdot r + 300\,\text{g} \cdot p + 200\,\text{g} \cdot \ell &\geq 700\,\text{g} \qquad \mid \div 100\,\text{g} \\ r + 3p + 2\ell &\geq 7\,, \end{aligned} \tag{3}$$

and dividing inequality (2) by 150 g results in

$$\begin{aligned} 750\,\text{g} \cdot r + 150\,\text{g} \cdot p + 450\,\text{g} \cdot \ell &\geq 4060\,\text{g} \qquad \mid \div 150\,\text{g} \\ 5r + p + 3\ell &\geq 27.06\,. \end{aligned} \tag{4}$$

Since we have to buy whole 1 kg packs of food, the variables r, p, and ℓ are integers, so $5r + p + 3\ell$ is an integer as well. Inequality (4) says that $5r + p + 3\ell \geq 27.06$, and the smallest integer that achieves this is 28, so we can strengthen inequality (4) to

$$5r + p + 3\ell \geq 28\,. \tag{5}$$

The sum of inequalities (3) and (5) is $6r + 4p + 5\ell \geq 35$, which multiplied by 10 kr is

$$\begin{aligned} 6r + 4p + 5\ell &\geq 35 \qquad \mid \cdot 10\,\text{kr} \\ 60\,\text{kr} \cdot r + 40\,\text{kr} \cdot p + 50\,\text{kr} \cdot \ell &\geq 350\,\text{kr}\,, \end{aligned} \tag{6}$$

which says that we have to spend at least 350 kr to get enough food for the current prices given in Table 1. These simple calculations, which can be verified using pen and paper, show that it is impossible to buy sufficient food for *less* than 350 kr. So if I tell you a solution that costs 350 kr, then you know that this solution is optimal.

While this is a very simple example, it shows that arguing about the optimality of a solution can be quite tricky. Modern automated tools can deal with far more complicated problems, where proving that the answers are correct can be vastly more difficult, but this thesis illustrates how state-of-the-art tools can be enhanced to output certificates that show that their result is correct. The approach presented in this thesis can be used for a wide range of different automated optimization tools to certify the correctness of their result.

Contribution Statement

The following papers are included in this thesis:

Paper I Stephan Gocht, Jakob Nordström, Ruben Martins, and Andy Oertel. "Certified CNF Translations for Pseudo-Boolean Solving". Accepted for publication in *Journal of Artificial Intelligence Research*. Preliminary version in *Proceedings of the 25th International Conference on Theory and Applications of Satisfiability Testing (SAT '22)*, volume 236 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 16:1–16:25, August 2022.

Paper II Jeremias Berg, Bart Bogaerts, Jakob Nordström, Andy Oertel, and Dieter Vandesande. "Certified Core-Guided MaxSAT Solving". In *Proceedings of the 29th International Conference on Automated Deduction (CADE-29)*, volume 14132 of *Lecture Notes in Computer Science*, pages 1–22. Springer, July 2023.

Paper III Jeremias Berg, Bart Bogaerts, Jakob Nordström, Andy Oertel, Tobias Paxian, and Dieter Vandesand. "Certifying Without Loss of Generality Reasoning in Solution-Improving Maximum Satisfiability". In *Proceedings of the 30th International Conference on Principles and Practice of Constraint Programming (CP '24)*, volume 307 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 4:1–4:28, September 2024.

Paper IV Alexander Hoen, Andy Oertel, Ambros Gleixner, and Jakob Nordström. "Certifying MIP-Based Presolve Reductions for 0–1 Integer Linear Programs". In *Proceedings of the 21st International Conference on the Integration of Constraint Programming, Artificial Intelligence, and Operations Research (CPAIOR '24)*, volume 14742 of *Lecture Notes in Computer Science*, pages 310–328. Springer, May 2024.

Paper V Stephan Gocht, Ciaran McCreesh, Magnus O. Myreen, Jakob Nordström, Andy Oertel, and Yong Kiam Tan. "End-to-End Verification for Subgraph Solving". In *Proceedings of the 38th AAAI Conference on Artificial Intelligence (AAAI '24)*, pages 8038–8047, Febuary 2024.

Paper VI Hannes Ihalainen, Andy Oertel, Yong Kiam Tan, Jeremias Berg, Matti Järvisalo, Magnus O. Myreen, and Jakob Nordström. "Certified MaxSAT Preprocessing". In *Proceedings of the 12th International Joint Conference on Automated Reasoning (IJCAR '24)*, volume 14739 of *Lecture Notes in Computer Science*, pages 396–418. Springer, July 2024.

Paper VII Markus Anders, Bart Bogaerts, Benjamin Bogø, Arthur Gontier, Wietze Koops, Ciaran McCreesh, Magnus O. Myreen, Jakob Nordström, Andy Oertel,

Adrian Rebola-Pardo, and Yong Kiam Tan. "Faster Certified Symmetry Breaking Using Orders With Auxiliary Variables". In *Proceedings of the 40th Annual AAAI Conference on Artificial Intelligence (AAAI '26)*, January 2026.

Paper VIII Ciaran McCreesh, Jakob Nordström, Andy Oertel, and Yong Kiam Tan. "Proof Logging for Projected Enumeration (and Counting?) Problems in VeriPB". *in review*.

There are additional papers that Andy Oertel contributed to, which are not included in this thesis:

- Emir Demirović, Ciaran McCreesh, Matthew J. McIlree, Jakob Nordström, Andy Oertel, and Konstantin Sidorov. "Pseudo-Boolean Reasoning About States and Transitions to Certify Dynamic Programming and Decision Diagram Algorithms". In *Proceedings of the 30th International Conference on Principles and Practice of Constraint Programming (CP '24)*, volume 307 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 9:1–9:21, September 2024.
- Wietze Koops, Daniel Le Berre, Magnus O. Myreen, Jakob Nordström, Andy Oertel, Yong Kiam Tan, and Marc Vinyals. "Practically Feasible Proof Logging for Pseudo-Boolean Optimization". In *Proceedings of the 31st International Conference on Principles and Practice of Constraint Programming (CP '25)*, volume 340 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 21:1–21:27, August 2025.
- Simon Dold, George Katsirelos, Wietze Koops, Magnus O. Myreen, Jakob Nordström, Andy Oertel, and Yong Kiam Tan. "End-to-End Certified Graph Colouring". *in review*
- Berhan Oumer Adame, Bart Bogaerts, Benjamin Bogø, Simon Dold, Arthur Gontier, Wietze Koops, Ciaran McCreesh, Matthew McIlree, Jakob Nordström, Andy Oertel, Adrian Rebola-Pardo, and Mark Turnbull. "Proof Trimming for Pseudo-Boolean Proofs with Strengthening Rules". *manuscript in progress*

For the included papers, the table below indicates the work preformed by Andy Oertel in the corresponding projects. There were also continuous discussions between all collaborators throughout the project and especially in early stages.

Paper	*Concept*	*Implementation*		*Evaluation*	*Writing*
		Verifier	*Solver*		
I	◑	○	◕	◔	◑
II	◕	●	◕	●	◕
III	◑	●	◔	●	◑
IV	◑	●	○	◑	◑
V	◑	◕	○	○	◔
VI	◑	◕	○	●	◑
VII	◑	◕	○	◔	◑
VIII	◑	◕	○	○	◔

The dark portion of the circle is intended to illustrate the proportion of work and responsibilities assigned to Andy Oertel, but is more formally defined as:

- ● Andy Oertel led and did almost all the work.
- ◕ Andy Oertel led and did a majority of the work.
- ◑ Andy Oertel was a contributor to the work.
- ◔ Andy Oertel was a minor contributor to the work.
- ○ Andy Oertel did not contribute, except for proofreading.

Concept Coming up with the ideas and working out the details in theory.

Implementation Writing the software required for the paper.

Evaluation Conducting the experimental evaluation and analysing the data.

Writing Drafting and editing the paper.

The following discusses the contributions by each author and especially the contribution by Andy Oertel in more detail.

Paper I

Andy Oertel developed the theory for certifying the binary adder encoding, with help from Jakob Nordström, and helped Stephan Gocht in developing the general framework to certify different encodings. Stephan Gocht provided prototype implementations for the sequential counter encoding and the (generalized) totalizer encoding. The final implementation was done by Andy Oertel with the help of Ruben Martins who implemented the certifying sequential counter encoding. The benchmark set and data to be measured in the experiments was discussed with all authors. Andy Oertel wrote the binary adder section of the paper, provided figures and examples to all sections of the paper and improved the evaluation section. All authors were involved in discussions about the structure of the paper and helped to polish the manuscript.

Paper II

Andy Oertel was the lead author of the paper. All authors were involved in discussions to develop the theoretical foundations to certify core-guided MaxSAT solvers. Andy Oertel mainly worked out the details to make certification feasible in practice. Andy Oertel and Dieter Vandesande implemented certification into CGSS. Dieter Vandesande implemented the certification for the incremental totalizer encodings and Andy Oertel implemented everything else. Andy Oertel implemented improvements in VeriPB to improve the proof checking performance. Andy Oertel conducted the experiments and analysed the experimental results. All authors were involved in discussing benchmark sets and data to be measured. Andy Oertel found the bug in the original implementation of CGSS and provided fixes for the bug. The structure of the paper was discussed with all authors. The preliminaries, the description of how to do certification for core-guided MaxSAT,

and the experimental evaluation were written by Andy Oertel, which were later improved by all authors.

Paper III

The theoretical idea to certify the dynamic generalized polynomial watchdog encoding was mainly developed by Bart Bogaerts in discussion with all authors. The first idea was discussed and improved through discussions with all authors. Jeremias Berg, Bart Bogaerts, and Andy Oertel discussed alternative approaches and found good reasons that explain why the approach proposed in the paper is required. Tobias Paxian and Dieter Vandesande implemented certification in the MaxSAT solver Pacose with assistance from Andy Oertel. Improvements and additional tracking of proof statistics in the proof checker were implemented by Andy Oertel. Andy Oertel conducted the experiments and analysed the experimental results. All authors were involved in discussing benchmark sets and data to be measured. The structure of the paper was discussed with all authors. Andy Oertel wrote the preliminaries on pseudo-Boolean proof logging and the experimental evaluation, which were later improved by all authors.

Paper IV

Most of the certification was developed by Ambros Gleixner, Alexander Hoen, and Jakob Nordström, while Andy Oertel helped to figure out certification for some remaining presolving techniques. The need for an objective update rule was discovered by Alexander Hoen and Andy Oertel. Andy Oertel developed the objective update rule and implemented it together with further checking improvements into the proof checker VeriPB. Alexander Hoen implemented certification in the MIP presolver PaPILO. Alexander Hoen and Andy Oertel jointly conducted the experiments and Alexander Hoen analysed the experimental data. The structure of the paper was discussed with all authors. Andy Oertel wrote the preliminaries on pseudo-Boolean proof logging and about the new objective update rule, and he helped Alexander Hoen in writing the description on the certification of presolving reductions.

Paper V

Andy Oertel joined this project after it was running, and the other authors developed to the original idea of the project. Andy Oertel worked out all elaboration algorithms in detail, which were discussed in meetings with Jakob Nordström. Andy Oertel improved the elaboration algorithm to achieve the necessary performance. The prototype implementation of the elaboration in VeriPB was done by Stephan Gocht. Andy Oertel changed major parts of this preliminary implementation to accommodate further improvements to the elaboration algorithms and support for all proof rules. Andy Oertel also implemented checked deletion into VeriPB. Magnus Myreen and Yong Kiam Tan implemented the formally verified proof checker CakePB, where Andy Oertel helped to formalize the correctness proof of the strengthening rules. The structure of the paper was discussed with

all authors. Andy Oertel wrote the proof elaboration section, which was later improved by all authors.

Paper VI

Certification for almost all MaxSAT preprocessing techniques was developed by Jeremias Berg, Hannes Ihalainen, and Matti Järvisalo. Jakob Nordström and Andy Oertel helped them to develop certification for the techniques of hardening and label matching. The proof format for problem reformulation proofs was jointly developed through discussions between all authors. Hannes Ihalainen implemented the certification into the MaxSAT preprocessor MaxPre. Support for proofs of problem reformulation was implemented by Andy Oertel into VeriPB and by Magnus Myreen and Yong Kiam Tan into CakePB. Andy Oertel conducted the experiments and analysed the experimental results. All authors were involved in discussing benchmark sets and data to be measured. The structure of the paper was discussed between all authors. Andy Oertel wrote the preliminaries and the experimental evaluation, which was later improved by all authors.

Paper VII

The concept of extending the certification system to allow for orders with auxiliary variables was developed jointly by all authors together. Details for making the proof format and implementation efficient were worked out by Wietze Koops and Andy Oertel. The certification was implemented in the symmetry breaking tool satsuma by Markus Anders and Wietze Koops. As part of this project, Andy Oertel reimplemented VeriPB and took the lead in implementing the support for orders with auxiliary variables, where Markus Anders, Benjamin Bogø, Arthur Gontier, Wietze Koops, and Adrian Rebola-Pardo made additional contributions. The support for orders with auxiliary variables in CakePB was implemented by Magnus Myreen and Yong Kiam Tan. All authors were involved in discussing benchmark sets and data to be measured. Ciaran McCreesh conducted the experiments and analysed the experimental results. The structure of the paper was discussed between all authors. Andy Oertel mainly wrote the introduction, preliminaries, checker implementation, concluding remarks, and parts of the appendix.

Paper VIII

The idea and concepts for proof logging projected enumeration and preserving projected model counts was developed in joint discussions with all authors. Andy Oertel worked out the details of the proof rules and how to keep track of the preserved variables in the proof system and checker. Andy Oertel implemented checking for enumeration proofs and projected model count preserving proofs into VeriPB and led the implementation to determine the exact format. Yong Kiam implement the necessary changes into the formally verified proof checker CakePB. The experimental evaluation and the implementation into the solver was performed by Ciaran McCreesh. All authors were involved in discussing benchmark sets. The

structure of the paper was discussed between all authors. Ciaran McCreesh took the lead in writing the paper, but all other authors contributed to the writing of the paper. Andy Oertel mainly wrote some preliminaries and technical details about the extensions of the proof system for enumeration.

Kappa

Certifying Combinatorial Optimization: A Unified Approach Using Pseudo-Boolean Reasoning

---

1 Introduction

Over the past decades, combinatorial optimization solvers have advanced vastly across all major paradigms, making it possible to solve increasingly larger and more complex optimization problems. As a result, such solvers have become important tools for both commercial and academic applications. For instance, Boolean satisfiability (SAT) and maximum Boolean satisfiability (MaxSAT) solving [BHvMW21] are used for hardware verification [BCCZ99], chip design [CNR21], or proving theorems [HKM16, SH23]. Constraint programming (CP) [RvBW06] is widely used for personnel allocation and timetabling [Wal96], power plant production planning [BBVC13], and sports league scheduling [Wei25]. Mixed integer programming (MIP) [AW13] is used for supply chain optimization [GGK+19], public transport planning [Sch20], and investment portfolio optimization [MOS15].

The improved performance of modern solvers is achieved through more sophisticated and specialized reasoning techniques that have to be integrated with each other. This, in turn, increases the implementation complexity of modern solvers. This complexity naturally raises the question if modern combinatorial solvers are implemented correctly. Correctness is especially crucial in mission-critical domains, where incorrect solutions can have severe real-world consequences, such as ambulance dispatch [AV07], kidney exchange programs [MO12], or air traffic control [HPRS24]. It is well-known that solvers across all paradigms and of different levels of maturity contain bugs and may return incorrect results [BLB10, CKSW13, AGJ+18, GSD19, GS19, BBN+23, PB23, WS24]. This makes it difficult to trust that newly developed, cutting-edge techniques are implemented correctly in solvers.

To mitigate the issue of incorrect solver results, the several approaches have been proposed in the software engineering literature. The most widely used approach in software engineering is *testing*, for which we verify if the output of a program matches the expected output for known input-output pairs [MSB11]. However, this approach is inherently limited to known input-output pairs, which can be generated manually or automatically, for example through *fuzzing* [MKL+95, ZWCX22, PB23].

Within combinatorial optimization, the most successful testing strategy is fuzzing, where inputs are generated such that we know the expected output and solvers have to use all their components [ABS13, PB23]. However, testing can only show the presence of faults, but cannot give correctness guarantees.

At the opposite end of the spectrum lies *formal verification*, where the program's behaviour is formally specified, and the implementation is rigorously proven to adhere to this specification [HT15]. This approach provides guarantees of correctness with respect to the specification, but the effort required to formally verify a modern combinatorial solver is enormous and only grows as solver get more complex. The most advanced example is probably the formally verified SAT solver IsaSAT [FL23], which performs far worse than other modern SAT solvers, even though SAT is a comparatively simple combinatorial optimization paradigm.

In this thesis, the focus is on the approach of *certifying algorithms* [MMNS11], which offers a good middle ground between testing and formal verification. The key idea is that an algorithm should not just return an output, but also produces a certificate that shows that the output is correct. The certificate should make it straightforward to verify that the returned output is correct for the given input. As a result, we do not need to trust the solver's implementation or internal reasoning, but it suffices to trust a separate, typically much simpler, checker that verifies the correctness of the solver output using the certificate. Such checkers are often simple enough to be formally verified. Therefore, this approach offers strong correctness guarantees for solver outputs, which is precisely the guarantee that matters in practice, since we just want to know that our problem was solved correctly.

Although the idea of certifying algorithms is already used by the extended Euclidean algorithm [MMNS11] and primal-dual optimization algorithms [Far02], dedicated research into certifying algorithms started with the LEDA project [MN89, MN95]. The term certifying algorithms was first used by Kratsch et al. [KMMS06]. In combinatorial optimization, certifying algorithms have gained increasing traction across several paradigms [BFT11, CGS17, VS10]. For instance, certifying algorithms are now standard for SAT solving, which was achieved mainly by requiring all solvers participating in the main track of the annual SAT competition to be certifying [SAT13]. This popularity of certifying algorithms led to many certification systems for SAT [Heu21], like RUP [GN03, Van08], DRAT [JHB12], LRAT [CMS17], GRAT [Lam20], FRAT [BCH21], PR [HKB17], and SR [BT21].

This thesis studies and extends a certification system based on pseudo-Boolean reasoning called VeriPB [BGMN23, GN21, Goc22]. VeriPB is inspired by the success of certification in SAT solving and is based on the cutting planes proof system from proof complexity [CCT87]. This thesis extends the VeriPB system from decision problems to optimization and enumeration problems by introducing new rules that capture the reasoning of optimization and enumeration solvers, enabling the certification of bounds on the optimal value and enumerations of solutions. Furthermore, we show how VeriPB can certify problem reformulations independently of solving, providing various formal guarantees how the reformulated problem relates to the original problem. Finally, to provide formal correctness guarantees for solver outputs, we introduce CakePB, a framework for formally verified checkers that makes it straightforward to obtain formally verified checkers for different combinatorial optimization paradigms.

Overall, this thesis makes progress towards a general certification framework for combinatorial optimization using one unified multipurpose system. Specifically, we demonstrate that VeriPB can be used to add certification to a variety of algorithms to solve MaxSAT, subgraph, 0-1 integer linear programming, graph colouring, and dynamic programming problems. In related work, VeriPB has also been used to certify advanced SAT solving techniques [GN21, BGMN23], constraint programming [EGMN20, GMN22, MM23, MMN24, MM25], and automated planning [DHN+25]. Moreover, extending VeriPB to fully support mixed integer programming has been proposed [DEGH23], which would directly enable certification for many more combinatorial optimization problems.

The first part of this thesis is a comprehensive summary of the work (the so-called *cover essay* or *kappa*), which is structured as follows. Section 2 introduces background for the topics discussed in this thesis, which includes a review of different combinatorial optimization paradigms and an introduction to certifying algorithm. We review related work about certifying algorithms for combinatorial optimization in Section 3. Section 4 presents the pseudo-Boolean certification system that we use to certify combinatorial optimization algorithms. This section provides a full description of the certification system including all extensions from this thesis. We discuss the contributions of this thesis in detail in Section 5. The summary of the thesis ends with some concluding remarks and future work in Section 6. The second part of this thesis consists of the included papers.

2 Background

This section introduces the necessary preliminaries required to understand this thesis together with the used notation. It is assumed the reader has basic knowledge in Theoretical Computer Science, including basic computational complexity, logic, and graph theory. For additional background on computational complexity and logic see [AB16] and for some background on graph theory see [Die16].

2.1 Basic Notation

It follows some review of standard logic notation, which can be found, e.g., in [AB16]. We use $\top$ to denote true (tautology) and $\bot$ to denote false (contradiction). The symbol $\wedge$ denotes a logical conjunction, $\vee$ a logical disjunction, $\Rightarrow$ a material implication, and $\Leftrightarrow$ a material equivalence. A *Boolean variable* is a variable with domain $\{\bot, \top\}$, where we associate $\bot$ with 0 and $\top$ with 1. A *literal* of a Boolean variable x is either the Boolean variable itself x or its *negation* $\overline{x}$ also denoted $\neg x$, which can be though of $\overline{x} = 1 - x$.

2.2 Combinatorial Optimization

We will start this section with a review of standard notation and definitions that will be used throughout this thesis, and that can be found in, e.g., [Sau24, BN21]. For more history on combinatorial optimization, see [Sch05].

The goal of *mathematical optimization* is to find an optimal element in a set of feasible elements [Sau24]. A *combinatorial optimization* problem is a special case of mathematical optimization problem where the feasible elements are (at least partially) from a discrete set. While most of the definitions can be extended to arbitrary sets, in this thesis we mainly consider *Boolean optimization problems* (aka 0–1 *optimization problems*), i.e., the feasible elements are from a subset of $\{0,1\}^n$. The optimal value is usually defined by the minimum or maximum of an *objective* function $f : \{0,1\}^n \to \mathbb{Z}$. Without loss of generality, we can assume that f should be minimized, since we can consider minimizing $-f$ if f should be maximized. A *constraint* is a function $C : \{0,1\}^n \to \{\bot, \top\}$. The set of feasible elements is further restricted by a *formula* F, which is a set of constraints. An element is feasible if and only if all constraints in F evaluate to $\top$ on this element, i.e., the conjunction of all constraints in F is $\top$. A *trivial constraint* maps any input to $\top$ and a *contradictory constraint* maps any input to $\bot$. Therefore, we write a general combinatorial optimization problem as

$$\begin{aligned} \min \quad & f \\ \text{s.t.} \quad & F\,. \end{aligned}$$

Hence, the question in the popular science summary is the optimization problem

$$\begin{aligned} \min \quad & 60r + 40p + 50\ell \\ \text{s.t.} \quad & 100r + 300p + 200\ell \geq 700 \\ & 750r + 150p + 450\ell \geq 4060\,. \end{aligned}$$

A *decision problem* is a special case of an optimization problem where we are only interested in knowing if there is a feasible element with respect to the constraint. Hence, the objective function can be viewed as being constant, e.g., $f : \vec{x} \mapsto 0$.

Each dimension of the feasible set $\{0,1\}^n$ is associated with a Boolean variable. For a set of constraint F or an objective f, we use the notation $F(\vec{x})$ or $f(\vec{x})$ to stress that F or f is defined over the vector of Boolean variables $\vec{x} = x_1, \ldots, x_n$. For a formula $F(\vec{x})$, we can partition the vector of Boolean variables $\vec{x}$ by writing $F(\vec{y}, \vec{z})$ or $F(\vec{a}, \vec{b}, \vec{c})$ meaning $\vec{x} = (\vec{y}, \vec{z})$ or $\vec{x} = (\vec{a}, \vec{b}, \vec{c})$, respectively.

A *(partial) assignment* ρ is a (partial) function from variables to $\{\bot, \top\}$. A *substitution* ω is a generalization of an assignment where variables can also map to literals. Hence, we consider a (partial) assignment to be a special case of a substitution, where all unassigned variables map to themselves. Substitutions are extended to literals by defining for the negation of a variable that $\omega(\overline{x}) = \neg\omega(x)$, and to preserve truth values, i.e., $\omega(0) = 0$ and $\omega(1) = 1$. When denoting a substitution, then all variables that are not explicitly mentioned are mapped to themselves, e.g., the substitution $\{x \mapsto \overline{y}, z \mapsto 0\}$ maps x to $\overline{y}$, z to 0, and all other variables to themselves. The domain $\mathrm{dom}(\omega)$ of a substitution ω is the set of variables that are not mapped to themselves, e.g., the substitution $\{x \mapsto \overline{y}, z \mapsto 0\}$ has the domain $\{x, z\}$.

For a list of variables $\vec{x} = x_1, \ldots, x_n$ and a substitution ω, we define $\omega(\vec{x}) = \omega(x_1), \ldots, \omega(x_n)$. A substitution α can be *composed* with another substitution ω by applying ω first and then α, which we write as $(\alpha \circ \omega)(\vec{x}) = \alpha(\omega(\vec{x}))$. We can

apply a substitution ω to a constraint $C(\vec{x})$, which is denoted by $C(\vec{x})\restriction_{\omega}$ or $C\restriction_{\omega}$, by first applying ω on $\vec{x}$ and then evaluating C on $\omega(\vec{x})$, i.e., every variable x_i of C is substituted by $\omega(x_i)$. A substitution ω *satisfies* a constraint C if $C\restriction_{\omega}$ is trivial, i.e., $C\restriction_{\omega}$ is true regardless how the remaining variables are assigned, and ω *falsifies* C if $C\restriction_{\omega}$ is contradictory, i.e., $C\restriction_{\omega}$ is false regardless how the remaining variables are assigned.

There are many paradigms of combinatorial optimization that have been studied, where each paradigm restricts the combinatorial optimization problem in some way or takes a different view on how the constraints are formulated. Some key paradigms that are relevant for this thesis are Boolean satisfiability (SAT), maximum satisfiability (MaxSAT), and pseudo-Boolean optimization, which are introduced in Sections 2.2.1 to 2.2.3.

2.2.1 Boolean Satisfiability (SAT)

The *Boolean satisfiability (SAT) problem* is one of the core decision problems in computer science [BHvMW21] and is the canonical NP-complete problem [Coo71, Lev73]. To define the SAT problem we need some further notation. A *(disjunctive) clause* is a disjunction of literal, e.g., $x \lor \overline{y} \lor z$, which is the type of constraint considered for the SAT problem. A *Boolean formula* can always be written in *conjunctive normal form (CNF)*, which is a conjunction of clauses, e.g., $(x \lor \overline{y}) \land (\overline{x} \lor y) \land (\overline{x} \lor \overline{y})$. The constraints of a SAT problem are typically given in CNF.

The SAT problem is a decision problem that asks if there exists an assignment that satisfies a Boolean formula. If there exists such an assignment, then the formula is said to be *satisfiable*. Otherwise, the formula is said to be *unsatisfiable*.

The SAT problem can also be phrased in terms of a combinatorial decision problem. An element for in a SAT problem is an assignment, the constraints are the clauses, and we want to find an assignment that satisfies all clauses. We can also additionally associate the truth value $\bot$ with 0 and $\top$ with 1 so that the considered elements are in $\{0, 1\}^n$.

A core part of all modern SAT solvers is *conflict-driven clause learning (CDCL)*, which is enhanced with techniques for pre- and inprocessing as briefly mentioned in Section 2.2.4. We will briefly review the CDCL algorithm presented in Algorithm 1, where more detail about the components can be found in [MSLM21]. A key subroutine of the CDCL algorithm is *unit propagation*. Given a partial assignment ρ, a clause C unit propagates the literal ℓ if all literals except ℓ are mapped to $\bot$ by ρ in C, resulting in the assignment $\rho \circ \{\ell \mapsto \top\}$.

Example 1. To illustrate how unit propagation works, let us consider the clause $x \lor \overline{y} \lor \overline{z}$ and the partial assignment $\rho = \{x \mapsto \bot, y \mapsto \top\}$. All literals except $\overline{z}$ are false with respect to ρ. Therefore, the clause propagates $\overline{z}$ and the resulting partial assignment is $\{x \mapsto \bot, y \mapsto \top, z \mapsto \bot\}$.

The first step in the CDCL loop is to do unit propagation on the clauses in F starting with the current partial assignment ρ. If the propagated assignment ρ satisfies the formula, then we are done. Otherwise, we check if there is a clause that is falsified by ρ, which is called a *conflict*. If there is no such clause, we assign an unassigned variable to a value. If there is a conflict clause, we learn

Algorithm 1: Basic skeleton of the conflict-driven clause learning algorithm, which is the core algorithm of modern SAT solvers. The input is a Boolean formula F in CNF and the output is if F is satisfiable or not.

```
1  conflictDrivenClauseLearning(F):
2  ρ ← ∅;
3  Loop
4      ρ ← unitPropagation(F, ρ);
5      if ρ satisfies F then
6          return SAT;
7      if ρ falsifies a clause in F then
8          C ← conflictAnalysis(F, ρ);
9          if C is the empty clause then
10             return UNSAT;
11         F ← F ∪ C;
12         ρ ← backjump();
13     else
14         ρ ← decideVariable();
```

a new clause based on the propagations and decisions that were responsible to falsify the clause, which is called *conflict analysis*. If the learnt clause is the empty clause, then the formula is unsatisfiable, as the empty clause cannot be satisfied by any assignment. Otherwise, the learnt clause is added to the formula and some variables are unassigned, which is called *backjumping*. There is a lot more to modern SAT solving algorithms, but this summary describes the main aspects of the CDCL loop. The described algorithm is sound and complete [MS99], but we will not prove this here in the interest of space.

2.2.2 Maximum Satisfiability

The canonical extension of the SAT problem to an optimization problem is the *maximum satisfiability (MaxSAT)* problem [BJM21]. Given a set of (weighted) soft clauses and a set of hard clauses, the MaxSAT problem asks for an assignment that maximizes the sum of the weights of satisfied soft clauses subject to satisfying all hard clauses. In practice, the MaxSAT problem is more commonly formulated as finding an assignment that minimizes an integer linear function f over literals subject to satisfying all clauses, where $\bot$ and $\top$ are associated with 0 and 1, respectively. Without loss of generality, all coefficients in the objective function f are assumed to be positive by using that $\overline{x} = 1 - x$. The following paragraph explains how we translate a MaxSAT problem using soft and hard clauses into a MaxSAT problem using objective and constraints and vice versa.

Let S be the set of soft clauses and H be the set of hard clauses defining a MaxSAT problem. The set of constraints F is the union of all hard clauses and the clauses $b_i \vee C_i$ for each weighted soft clause $(w_i, C_i) \in S$ with weight w_i and soft clause C_i, where b_i is a variable that is not used in the hard and soft

Algorithm 2: Basic skeleton of the solution-improving search algorithm to solve the MaxSAT problem with objective f and Boolean formula F in CNF. The output is the optimal value of the MaxSAT problem, where the optimal value ∞ means that F is unsatisfiable.

```
1 solutionImprovingSearch(F, f):
2 v ← ∞;
3 Loop
4 │ (sat?, ρ) ← solveSAT(F);
5 │ if sat? = UNSAT then
6 │ └ return v;
7 │ v ← f(ρ);
8 └ F ← F ∪ asCNF(f ≤ v − 1);
```

clauses. The objective is $\sum_i w_i b_i$ for the weights w_i of the weighted soft clauses $(w_i, C_i) \in S$. Therefore, if a soft clause C_i is falsified by an assignment, then b_i has to be satisfied, which increases the objective value by w_i. A MaxSAT problem with constraints F and objective $f = \sum_i a_i \ell_i$ can be translated to the hard clauses F and the weighted soft clauses are $(a_i, \overline{\ell_i})$. Hence, if a literal ℓ_i is satisfied, which increases the objective value by a_i, then the soft clause $\overline{\ell_i}$ is falsified and incurs a cost of a_i. Both formulations have the same optimal value [BJM21].

Example 2. As an example for how this translation works, let us consider the MaxSAT problem with hard clauses $\{x \lor \overline{y}, y \lor \overline{z}\}$ and the weighted soft clauses $\{(5, \overline{x} \lor z), (7, \overline{x} \lor y)\}$. The translated MaxSAT problem is the combinatorial optimization problem

$$\begin{aligned} \min \quad & 5b_1 + 7b_2 \\ \text{s.t.} \quad & x \lor \overline{y} \\ & y \lor \overline{z} \\ & b_1 \lor \overline{x} \lor z \\ & b_2 \lor \overline{x} \lor y \,. \end{aligned}$$

This problem can again be translated back to a MaxSAT problem with hard clauses $\{x \lor \overline{y}, y \lor \overline{z}, b_1 \lor \overline{x} \lor z, b_2 \lor \overline{x} \lor y\}$ and weighted soft clauses $\{(5, \overline{b_1}), (7, \overline{b_2})\}$.

There are several state-of-the-art solving techniques for MaxSAT solving that are based on SAT solvers. The most straightforward algorithm to solve MaxSAT is *solution-improving search (SIS)* [ES06, PRB18], which is outlined in Algorithm 2. The idea is to solve the constraints using a SAT solver. If there is a solution, then we compute the objective value for this solution and add clauses to the constraints which only allow solutions with strictly better objective values. We repeat running the SAT solver and adding clauses to the constraints until the solver returns that the constraints are unsatisfiable, which means that the best solution found so far is the optimal value. There are various encodings to enforce strictly better solutions [War98, BB03, ES06, JMM15, PRB18]. An *incremental SAT*

Algorithm 3: Basic skeleton of the core-guided algorithm for solving the MaxSAT problem with objective f and Boolean formula F in CNF. The output is the optimal value of the MaxSAT problem, where the optimal value ∞ means that F is unsatisfiable.

```
1  coreGuidedSearch(F, f):
2  Loop
3      α ← {ℓ ↦ 0 | ℓ ∈ lits(f)};
4      (sat?, ρ, κ) ← solveWithAssumptionsSAT(F, α);
5      if sat? = UNSAT then
6          if κ = ∅ then
7              return ∞;
8          (F, f) ← reformulateProblem(F, f, κ);
9      else
10         return f(ρ);
```

solvers [ES03] reuses information from previous calls to the solver and can be called with so-called *assumptions*, which is a partial assignment that should be extended to a total assignment by the solver. Such a solver helps to speed up SIS MaxSAT solvers and allows efficient encodings that only change the used assumptions from one call to the next.

Another important MaxSAT paradigm, which heavily relies on incremental SAT solvers, is *core-guided* MaxSAT solving [MDM14, IBJ21]. Another feature of incremental SAT solvers is that if the assumptions cannot be extended to a complete solution, then the solver returns a so-called *core*, which is a subset of the literals $\{\ell_1, \ldots, \ell_n\}$ assigned by the assumptions to true. If the core is empty, then the constraints are unsatisfiable. A non-empty core $\{\ell_1, \ldots, \ell_n\}$ says that at least one literal in it should be falsified to obtain a satisfying assignment to the constraints, which is expressed by the clause $\overline{\ell}_1 \vee \cdots \vee \overline{\ell}_n$. We will focus on the state-of-the-art OLL algorithm [AKMS12, MDM14] to handle the cores, but there are other core-guide algorithms like PMRes [NB14]. A general skeleton of the core-guided algorithm to solve MaxSAT is outlined in Algorithm 3, where we use $\mathsf{lits}(f)$ to denote the set of literals in the objective.

We will briefly review the OLL algorithm in sufficient detail that is required for this thesis and refer to Andres et al. [AKMS12] and Morgado et al. [MDM14] for more details. The OLL algorithm first calls the SAT solver with the assumptions that set every literal in the objective to $\perp$, which is the partial assignment that leads to the smallest possible objective value. If SAT solver is able to extend these assumptions to a complete assignment that satisfies all constraints, then we found an optimal solution, as the assumptions enforce the smallest possible objective value that can be achieved with this objective. If the SAT solver is not able to do this, it will return a core clause C. We say that the *weight* $w(C, f)$ of a core C is the smallest coefficient of a literal in C in the objective f. We will introduce as many new variables $c_1, \ldots, c_n$ as there are literals in C and add clauses enforcing that $c_i \Leftrightarrow \sum_i \ell_i \geq i$ for $i = 1, \ldots, n$, i.e., c_i is true if at least i literals of C are true. There

is now an equivalence between ℓ_i literals and c_i variables, so that $\sum_i \ell_i = \sum_i c_i$, which can be used to substitute the expression $\sum_i w(C,f)\ell_i$ in f by $\sum_i w(C,f)c_i$ resulting in the reformulated objective f_{ref}. This process is then repeated with the reformulated problem.

There are many other approaches used to solve MaxSAT, which are not of interest for this thesis. Additional approaches to solve MaxSAT using incremental SAT solvers are *implicit hitting set (IHS)* search [DB13] or *branch and bound MaxSAT* solvers [AH14, LXC+21]. Furthermore, there are approaches that do not rely on SAT solvers at all to solve the MaxSAT problem like *integer-linear programming (ILP) solvers* [Ach07].

2.2.3 Pseudo-Boolean Optimization

A further generalization of SAT and MaxSAT is the *pseudo-Boolean optimization (PBO)* problem [RM21]. In pseudo-Boolean optimization the constraints are *pseudo-Boolean (PB) constraints*, which are integer-linear inequalities over literals and the objective function is an integer linear function over literals. Here we are again using that convention that $\bot$ and $\top$ are associated to 0 and 1, respectively, and that $\overline{x} = 1 - x$. Without loss of generality, we assume that constraints are given in normalized form $\sum_i a_i \ell_i \geq A$, where the coefficients a_i are non-negative integers, the right-hand side is an integer, and the literals ℓ_i are over distinct variables. In the literature the normalized form is often further restricted to only allow for non-negative A, i.e., if $A < 0$, then we set A to 0. The right-hand side A is also referred to as the *degree (of falsity)*. SAT is a special case of PB solving, as a clause $\bigvee_i \ell_i$ has the same semantics as the pseudo-Boolean constraint $\sum_i \ell_i \geq 1$.

Pseudo-Boolean optimization is equivalent to 0–1 ILP, where negative literals are turned into positive literals using that $\overline{x} = 1 - x$. Hence, any ILP solver [Ach07] can be used to solve PBO. However, there are also specialized PB solvers that follow the idea of CDCL presented in Algorithm 1, which gives rise to solve the PB decision problem [LP10, EN18]. Pseudo-Boolean optimization can then be solved using the MaxSAT approaches discussed in Section 2.2.2 using an incremental PB decision solver instead of a SAT solver [DGD+21].

The only components that need to be changed to use Algorithm 1 to solve a pseudo-Boolean problem are the procedures for unit propagation, conflict analysis, and backjumping. We will focus on how unit propagation is implemented in pseudo-Boolean solvers, as this is also used in the pseudo-Boolean proof checker VeriPB developed as part of this thesis. To see that we can efficiently detect propagations of pseudo-Boolean constraints, we define the *slack* of a PB constraint, which measures how close a constraint is to be falsified by an assignment. The slack of a constraint $C \doteq \sum_i a_i \ell_i \geq A$ under the assignment ρ is $\mathit{slack}(C,\rho) := \sum_{\rho(\ell_i) \neq 0} a_i - A$, where we use $\doteq$ to denote *syntactic equivalence*. If $\mathit{slack}(C,\rho) < 0$, then C is falsified by ρ, as even setting all literal in C that are unassigned by ρ to 1 would not satisfy C. A PB constraint C containing $a_i \ell_i$ as a term propagates ℓ_i to 1 under an assignment ρ if and only if $\mathit{slack}(C,\rho) < a_i$, as setting ℓ_i to 0 would result in $\mathit{slack}(C, \rho \circ \{\ell_i \mapsto 0\}) < 0$, which says that C is falsified by $\rho \circ \{\ell_i \mapsto 0\}$. Implementing efficient propagation of pseudo-Boolean constraints is an active research area [Dev20b, NORZ24].

Example 3. To see how unit propagation for pseudo-Boolean constraints works, we consider the constraints

$$C \doteq 3x + 2y + z \geq 3 \tag{1}$$
$$D \doteq 5\overline{x} + 2\overline{y} + 2\overline{z} \geq 7\,. \tag{2}$$

We start with the empty assignment $\rho_0 = \emptyset$. The slack of constraint D is $\mathit{slack}(D, \rho_0) = 2$, hence D propagates $\overline{x}$ to 1, resulting in the new assignment $\rho_1 = \{x \mapsto 0\}$. The slack of C is now $\mathit{slack}(C, \rho_1) = 0$, hence C propagates both y and z to 1, resulting in the assignment $\rho_2 = \{x \mapsto 0, y \mapsto 1, z \mapsto 1\}$. Now D is falsified by ρ_2, since $\mathit{slack}(D, \rho_2) = -2$, which is a conflict.

Another approach to solver pseudo-Boolean optimization problems is to encode the PB constraints into clauses and then use a SAT solver to solve the constraints [ES06, MML14, SN15]. There are many encodings with different properties that are used to encode PB constraints into CNF [Bat68, War98, BB03, ES06, JMM15, PRB18]. These solvers use standard SAT solvers and directly benefit from any improvement for SAT solving.

2.2.4 Preprocessing

When solving combinatorial optimization problems in practice, many solvers first use some algorithm to reformulate the problem before using the main solving procedure. This approach of reformulating the problem is called *preprocessing* in the SAT community and *presolving* in the operations research community. It is also possible to reformulate the current problem maintained by the solver during the main solving procedure, which is called *inprocessing*. It has been shown for all kinds of combinatorial optimization paradigms that preprocessing is an important technique [ABG+20, IBJ22, HGH23]. This section only gives a brief overview over preprocessing and will not go into detail about specific techniques used in preprocessing, for which the reader is instead referred to Achterberg et al. [ABG+20], Berg et al. [BJK21], and Ihalainen et al. [IBJ22].

Preprocessing techniques can be grouped into two categories [ABG+20]. The first category are so-called *primal preprocessing* techniques, which preserve the set of feasible solution and only change how this set is described. The second category of techniques can change the feasible set if it is guaranteed that the optimal value stays unchanged, which are called *dual preprocessing* techniques.

Symmetry breaking is a dual preprocessing techniques, which restricts the set of feasible solutions to only contain a few feasible solutions from the symmetric set of solutions by introducing new constraints [Sak21]. Symmetry breaking is commonly only for *syntactic* symmetries of the constraints, where a syntactic symmetry for a Boolean optimization problem is a permutation of the literals mapping a set of constraints to itself. In particular, syntactically means that a symmetry σ is a substitution mapping variables to literals such that for every constraint $C \in F$ it holds that $C\!\upharpoonright_\sigma \in F$, which is easy to check. Hence, for a feasible solution ρ, we can obtain a new solution by composing the symmetry with the solution $\sigma \circ \rho$. By applying this construction iteratively on the resulting solution, we obtain a set of symmetric solutions. By adding new constraints, we

remove symmetric solutions as long as at least one solution in the set of symmetric solutions is preserved. To simplify the check that at least one solution is preserved, we order the set of symmetric solution and only check that at least one minimal solution in this set is preserved, which is sometimes called a *canonical* solution.

2.2.5 Enumeration and Counting Problems

Given an objective and a set of constraints, another question might be to *enumerate* all (optimal) feasible solution, which is to list all (optimal) solutions [Nie00, EPRL12, SIY$^+$20]. It might even be of interest to enumerate solution in order of increasing objective value [Mur68]. We only focus on enumerating feasible solutions. Optimal solutions can be enumerated by finding the optimal value, adding a constraint excluding all non-optimal solutions, and then enumerating feasible solutions of the resulting problem. Sometimes we are only interested into different solutions for a subset of *preserved* variables, which is referred to as *projected enumeration* [GKS09]. An assignment ρ is different with respect to a set of preserved variables P if the assignment projected to the variables in P is different. In other words, we are only interested into the assignment to the variables in P For example, if $P = \{x\}$, then the solutions $\{x \mapsto 1, y \mapsto 0\}$ and $\{x \mapsto 1, y \mapsto 1\}$ projected to P are both $\{x \mapsto 1\}$, hence the solutions are the same projected solution with respect to P.

Instead of listing all (optimal) feasible solutions explicitly, we might just be interested in how many (optimal) feasible solutions a combinatorial optimization problem has, which is a so-called *counting* problem [Rys63, Dar04, GSS21]. Since we do not need to go through all solutions explicitly, algorithms for counting can be more efficient than for enumeration the problem of counting solutions is in the complexity class #P [AB16]. Similar to enumeration, we can restrict the problem to only count solutions with respect to a preserved set, which is referred to as *projected counting* [ACMS15].

2.3 Proof Complexity and Proof Systems

The research area of *proof complexity* studies how efficiently reasoning systems can prove statements [Kra19]. A key concept of proof complexity are *proof systems* as defined by Cook and Reckhow [CR79].

A refutation proof system is a set of inference rules to derive new constraints. A *proof* is a sequence of inference rule applications that start with the original constraints describing the set of feasible solutions and each inference rule application adds a new constraint. A proof system must satisfy the following three conditions to be a proof system in the sense of Cook and Reckhow [CR79]:

Soundness: If there is a feasible solution, then the proof system can not show that there is no feasible solution.

Completeness: If there is no feasible solution, then there exists a proof in the proof system showing that there is no feasible solution.

Polynomial-time checkable: Each inference rule application can be checked in polynomial time in the size of the proof.

By using the soundness property, we can show that there were no feasible solutions for a problem if we can derive a constraint that obviously states that is a contradictory constraint. This type of proof system is called a *sequential refutation proof system*. Since we will not use any other types of proof systems in this thesis, we will refer to *sequential refutation proof systems* just as *proof systems*.

To denote the inference rules in a proof system, we will use the notation

$$\frac{H_1 \qquad \dots \qquad H_n}{C} \text{ Inference rule}$$

to say that the *conclusion* C can be derived if the *hypotheses* $H_1, \dots, H_n$ have been derived before or are part of the original formula. All inference rules that we consider in this section are polynomial-time checkable, since syntactic checks of the hypotheses are sufficient. We will now introduce a few proof systems that are relevant for this thesis.

The *resolution* proof system [Bla37, DP60, DLL62, Rob65] operates over clauses, i.e., the conclusion and hypotheses are all clauses. This means that if we manage to derive the empty clause, then this shows that the original set of clauses was unsatisfiable. The only inference rule in the resolution proof system is the *resolution rule*

$$\frac{C \vee x \qquad D \vee \overline{x}}{C \vee D} \text{ Resolution over } x$$

deriving the clause $C \vee D$ from the clauses $C \vee x$ and $D \vee \overline{x}$. This rule is sound, since any assignment satisfying $C \vee x$ and $D \vee \overline{x}$ has to satisfy C if $x = \bot$ or D if $x = \top$. For a proof that the resolution proof system is complete, see [Rob65].

Example 4. As an example for the resolution rule, let consider the clauses $x \vee y$ and $x \vee \overline{y} \vee z$, which can be resolved over y to obtain $x \vee z$.

The *cutting planes* proof system [CCT87] uses pseudo-Boolean constraints. This means that if we derive the constraint $0 \geq 1$, then the original set of pseudo-Boolean constraints was unsatisfiable. For additional details with of the cutting planes proof system, we refer the reader to [BN21]. We will use the notation $F \vdash C$ to say that there is a cutting planes derivation of the constraint C from the constraints in F, and write $F \vdash F'$ if $F \vdash D$ for each $D \in F'$. The cutting planes proof system has the following inference rules. The *literal axiom rule* for any literal ℓ is

$$\frac{}{\ell \geq 0} \text{ Literal axiom for } \ell \text{ ,}$$

which states that the constraint $\ell \geq 0$ can always be derived. This rule is sound, since ℓ either evaluates to 0 or 1. Thus, the constraint $\ell \geq 0$ is trivial.

Two pseudo-Boolean constraints can be added together using the *addition rule*

$$\frac{\sum_i a_i \ell_i \geq A \qquad \sum_i b_i \ell_i \geq B}{\sum_i (a_i + b_i)\ell_i \geq A + B} \text{ Addition} \cdot$$

The addition rule is sound, since the sums of the coefficients of the satisfied literals by an assignment ρ satisfying both constraints $\sum_{\rho(\ell_i)=1} a_i$ and $\sum_{\rho(\ell_i)=1} b_i$ are larger than A and B, respectively. Hence, ρ also satisfies the sum of the constraints, as $\sum_{\rho(\ell_i)=1} (a_i + b_i)$ is larger than $A + B$.

Example 5. Let us consider the constraints $x + 2y + 3z \geq 3$ and $2\overline{x} + 2y \geq 2$. The sum of both constraints is $x + 2\overline{x} + 4y + 3z \geq 5$, which is normalized using $\overline{x} = 1 - x$ to $\overline{x} + 4y + 3z \geq 4$.

A pseudo-Boolean constraint can be multiplied by a positive integer using the *multiplication rule*

$$\frac{\sum_i a_i \ell_i \geq A}{\sum_i m a_i \ell_i \geq mA} \text{ Multiplication by } m \in \mathbb{N}\,.$$

Multiplication is sound, since the degree and the sum of the coefficients for the satisfied literals are multiplied by the same integer.

Example 6. Multiplying the constraint $x+2y+3z \geq 3$ by 3 results in $3x+6y+9z \geq 9$.

A pseudo-Boolean constraint in normalized form can be divided by a positive integer d if all coefficients are divisible by d using the *specialized division rule*

$$\frac{\sum_i d a_i \ell_i \geq A}{\sum_i a_i \ell_i \geq \lceil A/d \rceil} \text{ Division of normalized constraint by } d \in \mathbb{N}\,.$$

The division rule is sound, as the sum of the coefficients of the satisfied literals by an assignment is divided by d and so is the degree divided by d without considering rounding up. However, since all coefficients are integers, the satisfiability of the constraint does not change if the degree is rounded to the next biggest integer. To allow for coefficient that are not divisible by d we consider the *(general) division rule*

$$\frac{\sum_i a_i \ell_i \geq A}{\sum_i \lceil a_i/d \rceil \ell_i \geq \lceil A/d \rceil} \text{ Division of normalized constraint by } d \in \mathbb{N}\,.$$

This rule is sound, since we can increase all coefficients that are not divisible by d by adding literal axioms before applying the specialized division rule.

Example 7. Dividing the constraint $3x + 4y + 8z \geq 7$ by 3 using the general division rule results in the constraint $x + 2y + 3z \geq 3$.

The rules so far are already sufficient to get a cutting planes proof system that is complete, since it can simulate resolution rule by adding the two hypotheses together and dividing by 2. Another inference rule that can be applied to normalized constraints is the *saturation rule* [DG02]

$$\frac{\sum_i a_i \ell_i \geq A}{\sum_i \min\{a_i, A\} \ell_i \geq A} \text{ Saturation of normalized constraint}\,\cdot$$

This rule is sound, as a constraint is satisfied if a literal is satisfied that has a coefficient of at least A. Gocht et al. [GNY19] showed there are derivations where either the division rule or the saturation rule requires fewer rule applications than the other rule. Therefore, we consider cutting planes with both the division and saturation rule to use the strengths of both rules.

Example 8. Saturating the constraint $x + 3y + 5z \geq 3$ results in $x + 3y + 3z \geq 3$.

Finally, we discuss *extension rules* that allow to derive constraints with new variables. The resolution proof system can be modified to the *extended resolution* proof system [Tse68] by adding the extension rule

$$\frac{q \notin \mathit{Vars}(F)}{\overline{\ell_1} \vee q \quad \overline{\ell_2} \vee q \quad \ell_1 \vee \ell_2 \vee \overline{q}} \text{ Extended resolution for new variable } q \,,$$

where $\mathit{Vars}(F)$ is the set of variables used by the original and derived constraints before this rule is applied. The extension rule introduces a new *extension variable* q and clauses forcing q to be true if and only if $\ell_1 \vee \ell_2$ is true. To introduce extension variables in an *extended cutting planes* proof we can use the *reification rule*

$$\frac{q \notin \mathit{Vars}(F)}{q \Rightarrow C \wedge q \Leftarrow C} \text{ Reification for new variable } q \,,$$

where $C \doteq \sum_i a_i \ell_i \geq A$ is any pseudo-Boolean constraint not containing the variable q, $q \Rightarrow C$ is the pseudo-Boolean constraint $A\overline{q} + \sum_i a_i \ell_i \geq A$, and $q \Leftarrow C$ is the constraint $(\sum_i a_i - A + 1)q + \sum_i a_i \overline{\ell}_i \geq \sum_i a_i - A + 1$. The constraint $q \Rightarrow C$ forces the constraint C to true if q is true and $q \Leftarrow C$ forces q to true if C is true.

2.4 Certifying Algorithms

This section briefly motivates and defines the approach of certifying algorithms. For more history, details, and examples for certifying algorithms we refer the reader to a survey by McConnell et al. [MMNS11].

We consider a program to be correct if it returns a correct output for a given input with respect to a formal specification of the function computed by the program. This means that an error in the specification is not considered incorrect. The problem that certifying algorithms is trying to solve is to know if a software program is correct. The approach of certifying algorithms offers a middle ground between testing [MSB11] and formal verification [HT15] by providing correctness guarantees for the specific input we consider, but with less effort than what is required for formal verification. The idea of certifying algorithms is something we are all familiar with from solving school maths problems. For instance, when solving equations, a way to make sure that the calculated result is correct is to plug in the calculated values for the variables in the equations and to check if all equalities hold. I.e., there is a procedure independent of the solving process to check the correctness of the result. However, as we have seen in the popular science summary, it is vastly more challenging to independently verify that no solution exists, which is the main focus of this thesis.

Before we can define what a certifying algorithm is, we require the following two definitions. The *precondition* of a function is the restriction on the input for which the function is valid. E.g., consider the function $\mathit{div}(a, b)$ for $a, b \in \mathbb{R}$ which divides a by b. The precondition of $\mathit{div}(a, b)$ would be that $b \neq 0$, as division by 0 is undefined. The precondition can also be trivially satisfied, if the function is defined for any value of the function domain. The *postcondition* of a function is the expected output of the function with respect to the input of the function. E.g., for the function $\mathit{div}(a, b)$ the postcondition would be that value of the function

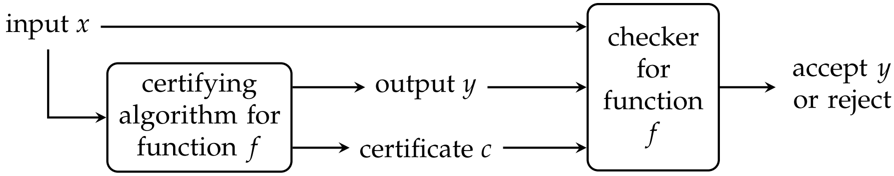


Figure 1: *Workflow for a certifying algorithm for the function f and checking the certificate with a checker for the function f.*

$div(a, b)$ is actually a/b, which can be verified by computing $a = b \cdot c$ where c is the computed result of $div(a, b)$.

A certifying algorithm get an input $x \in X$ and returns an output $y \in Y$ together with a *certificate*[2] c. The input to the checker are x, y, and c, which either verifies that the y is the correct output for x or fails to verify the correctness of y. In the latter case it does not necessarily mean that y is incorrect. It can be that y is correct for input x, but the certificate c does not show that y is correct, which means that there is an issue with generating the certificate. It can be the case that the certificate is trivial (empty) if the checker can verify the correctness of output y for input x without additional information. See Figure 1 for the workflow of a certifying algorithm and the verification of the certificate.

McConnell et al. [MMNS11] define three categories of certifying algorithms. All definitions consider algorithms that compute the function $f : X \to Y$. A *strongly certifying algorithms* halts for all inputs $x \in X$ and either returns that x does not satisfy the precondition, which is proven by the certificate, or returns $y \in Y$ and the certificate shows that $f(x) = y$. An *(ordinary) certifying algorithms* halts for all inputs $x \in X$ and either returns that x does not satisfy the precondition, which is proven by the certificate, or returns $y \in Y$ and the certificate shows that $f(x) = y$ if x satisfies the precondition. In the latter case it can happen that if x does not satisfy the precondition, then $f(x) \neq y$. A *weakly certifying algorithms* only has to halt for x satisfying the precondition. If the algorithm halts, then it either returns that x does not satisfy the precondition, which is proven by the certificate, or return $y \in Y$ and the certificate shows that $f(x) = y$ if x satisfies the precondition. Hence, if a weakly certifying algorithm halts, then it behaves exactly as an ordinary certifying algorithm.

To illustrate these different categories, we consider the above example of division. A strongly certifying algorithm for this problem would always halt and if the divider is 0, then the algorithm would return an error and a trivial certificate indicating that the divider is 0. An (ordinary) certifying algorithm for division might act as the strongly certifying algorithm and additionally is allowed to return anything as long as the certificate is correct. For instance, if we want to divide 0 by 0 and the algorithm outputs 0 the certificate check would still be correct, as $0 = 0 \cdot 0$. A weakly certifying algorithm is additionally allowed to run without halting if we divide by 0.

For the rest of this thesis, we will only consider the case that the precondition is trivial (i.e., it is always satisfied). If the precondition is trivial, then all certifying

[2] The certificate is also referred to as the *witness* by McConnell et al. [MMNS11].

algorithms are strongly certifying, as the exceptions for the other types can only occur when the precondition is not satisfied. In fact, McConnell et al. [MMNS11, Theorem 5] have shown that any deterministic algorithm with a trivial precondition has a strongly certifying algorithm for the same problem with the same asymptotic running time. Hence, all the algorithms we consider from now on are strongly certifying. In this thesis, we only consider deterministic algorithms that are guaranteed to terminate in this thesis with a trivial precondition for solving combinatorial optimization problems. Using the result by McConnell et al. [MMNS11, Theorem 5], we know that for any algorithm we consider in this thesis there exists an equivalent strongly certifying algorithm with a constant factor overhead in running time. Therefore, our goal is to only incur a constant overhead for generating the certificate. An example of a combinatorial optimization algorithm that is not deterministic is local search.

As the definitions by McConnell et al. for certifying algorithms only give guarantees about the algorithm, we can not get any guarantees about the implementation of the algorithm as software running on hardware. If we have bugs in the implementation, then we do not have any guarantees about the output or the certificate. Moreover, it can happen that we run into resource limits (e.g., not enough free memory) and do not produce a certificate or output at all. However, if an algorithm has been implemented correctly and enough resources are available, then the guarantees for this algorithm transfer to the practical implementation.

On the one hand, if the implementation returns an output and a certificate and the checker verifies that the output is correct, then the guarantees from the definitions for certifying algorithms hold. So for a strongly certifying algorithm we know that if the input satisfies the precondition, then the output satisfies the postcondition, and otherwise, the implementation correctly detected that the precondition is not satisfied.

On the other hand, if the checker rejects the output using the certificate, then there could be multiple reasons why this is the case. For instance possible reasons could be that the implementation computed an incorrect output, the implementation computed an incorrect certificate for a correct output, the implementation was prematurely terminated, or the implementation of the checker has a bug. Hence, the checker can reject a correct output, which is undesirable but also unavoidable.

Even if the checker rejects, the certification process can be useful to detect the problem. A checker can be designed in a way that it not just accepts the output of rejects. It can give a reason why it rejected, which can aid in the process of debugging where the implementation of the certifying algorithm went wrong.

2.4.1 Certifying Algorithms for SAT

The certificate for solving a SAT problem is different, depending on whether the formula is satisfiable or not. If the algorithm outputs satisfiable, then the certificate is a solution that satisfies the formula, where a partial assignment that trivializes the formula is sufficient. If the algorithm outputs unsatisfiable, then the certificate is a proof showing that it is impossible to satisfy the formula. As, this direction is the interesting case, certification in the SAT community is commonly referred to as *proof logging*. Many formats have been proposed in the last couple of decades,

which are discussed in detail in [Heu21]. We will briefly review different proof formats of unsatisfiability in chronological order.

Van Gelder [VG02] proposed to certify unsatisfiability by a *resolution proof* by using the resolution proof system as defined in Section 2.3. The resolution proof can be extracted from the clause learning procedure of CDCL.

Goldberg and Novikov [GN03] instead suggested a proof format based on so-called *reverse unit propagation (RUP)*. The assignment ρ obtained at the end of unit propagation has the property that assigning an assigned variable to the opposite value in ρ falsifies a clause in the formula. Hence, if an assignment obtained by unit propagation falsifies a clause, then the original assignment can never be extended to a satisfying assignment of the formula, as changing the assignment to satisfy this clause will falsify another clause. Reverse unit propagation shows that a clause $C \doteq \ell_1 \vee \cdots \vee \ell_k$ is implied by the formula by doing unit propagation that starts with an assignment that satisfies the negated clause $\neg C \doteq \overline{\ell}_1 \wedge \cdots \wedge \overline{\ell}_k$ and has to result in a conflict. This conflict shows that the only way the formula could be satisfied is if $\ell_1 \vee \cdots \vee \ell_k \doteq C$, which shows that we can add the clause C to the formula. To denote that $F \cup \{\neg C\}$ unit propagates to conflict, we use $F \vdash_1 C$. For sets of constraint F', the notation $F \vdash_1 F'$ means that $F \vdash_1 D$ for each $D \in F'$. Using this notation we can formally state that the constraint C can be derived by reverse unit propagation from the formula F if

$$F \vdash_1 C\,. \tag{3}$$

Example 9. The clauses $x \vee y$ and $x \vee \overline{y}$ imply the clause x by RUP, since the negation of x propagates x to false resulting in the assignment $\rho_1 = \{x \mapsto \bot\}$. The clause $x \vee y$ propagates y to true under ρ_1 resulting in $\rho_2 = \{x \mapsto \bot, y \mapsto \top\}$. The clause $x \vee \overline{y}$ is falsified by ρ_2, which successfully concludes the RUP check.

The performance of unit propagation depends on the number of clauses we know, as we do not know which clauses propagate. If we never delete clauses, then the performance of checking can deteriorate the more clauses we derive, which motivated the development of proof systems with deletions [HHW14]. Alternatively, performance can be improved by providing hints which constraints propagate in which order [CHH⁺17]. A common convention is to prefix systems that allow to delete constraints to prefix deletion (D), e.g., reverse unit propagation (RUP) with deletion becomes deletion reverse unit propagation (DRUP). All systems discussed in this section can be extended with a deletion rule.

While RUP can certify CDCL, dual pre- and inprocessing techniques can not be certified using RUP. The main issue is that RUP can only derive implied clauses, i.e., clauses that do not change the set of solutions to a formula. To certify pre- and inprocessing that derived non-implied clauses, Järvisalo et al. [JHB12] proposed proofs based on the *resolution asymmetric tautology (RAT)* rule, which guarantees satisfiability-equivalence of the formula and the formula with the added clause. The idea of the rule is to extend RUP[3] with one step of resolution. A clause C can be added to a formula F by RAT if there is a literal $\ell \in C$ such that the resolvent of C and any clause $C' \in F$ where $\overline{\ell} \in C'$ is RUP. The correctness of this rule can

[3]The property of *asymmetric tautology (AT)* is equivalent to RUP.

be seen by considering the case that C is not satisfied by some solutions to F, and specifically let us consider the solutions assigning ℓ to false. If we change the assignment of ℓ in these assignments to $true$, then C will be satisfied, but there might be clauses in F that are now falsified by this assignment. Hence, we have to check for all clauses $C' \in F$ containing the literal $\overline{\ell}$ that there is a solution satisfying the formula that also satisfies $C' \setminus \{\overline{\ell}\}$, which means that $C' \setminus \{\overline{\ell}\}$ is implied by F.

Alternatively, this can be formalized by saying that clause C with literal $\ell \in C$ can be derived by RAT from a formula F if

$$F \wedge \neg C \vdash_1 F\restriction_{\{\ell \mapsto \top\}} . \tag{4}$$

Here, the substitution $\{\ell \mapsto \top\}$ is called the *witness* and has to be specified explicitly. As the *proof obligations* of the rule, which are the clauses on the right-hand side of the implication, depend on the current formula, allowing to delete constraints from the formula can actually strengthen this proof rule [BT21]. While this rule does not preserve solutions with respect to propositional logic, it preserves solutions with respect to so-called *overwrite logic* [RS18].

Example 10. The clause x can be derived by RAT over the literal x from the clauses $x \vee y$ and $\overline{x} \vee y$. The negation of x propagates x to false, which makes $x \vee y$ propagate y to true, resulting in the assignment $\rho_1 = \{x \mapsto \bot, y \mapsto \top\}$. By the RAT condition for $\overline{x} \vee y$, the clause y is RUP, y is falsified by ρ_1.

RAT can be generalized to the *propagation redundancy (PR)* rule [HKB17][4], where the witness can be an arbitrary assignment. The idea for this rule is that adding a clause C could remove solutions (if there are any) for F, but there should be at least one solution to the formula that can be extended from the assignment ρ. Hence, we have to show that $C\restriction_\rho$ and for all $D \in F$ that $D\restriction_\rho$ is implied by the formula F. We can think of the witness as the way to repair any potential solution that could have been removed. Formally, the clause C can be derived by the PR rule from a formula F given a (partial) assignment ρ if

$$F \wedge \neg C \vdash_1 (\{C\} \cup F)\restriction_\rho . \tag{5}$$

Example 11. The clause x can be derived by PR using the assignment $\rho = \{x \mapsto \top, y \mapsto \top\}$ from the clause $\overline{x} \vee y$, since ρ satisfies x and $\overline{x} \vee y$.

The PR rule can be generalized even further to the *substitution redundancy (SR)* rule [BT21] by allowing a substitution as the witness. Similar to the PR rule, the SR rule uses the witness to repair solutions falsified by the added clause C. Formally, the clause C can be derived by SR from a formula F given a substitution ω if

$$F \wedge \neg C \vdash_1 (\{C\} \cup F)\restriction_\omega . \tag{6}$$

Example 12. The clause x can be derived by substitution redundancy using the substitution $\omega = \{x \mapsto y, y \mapsto x\}$ from the clause $x \vee y$ and $\overline{x} \vee \overline{y}$. The negation of x propagates x to 0, which makes $x \vee y$ propagate y to 1 resulting in the assignment

[4]Redundancy in the SAT community means that adding a clause to or removing a clause from a formula does not change the satisfiability of the formula.

$\rho_1 = \{x \mapsto 0, y \mapsto 1\}$. The clause x is substituted to y, which is RUP, since the negation of y propagates y to 0, which is contradicting ρ_1. The clause $x \vee y$ is substituted also $x \vee y$, which is RUP, since the negation of $x \vee y$ propagates y to 0. The clause $\overline{x} \vee \overline{y}$ is substituted also $\overline{x} \vee \overline{y}$, which is RUP, since the negation of $\overline{x} \vee \overline{y}$ propagates x to 1. This shows that x follows by substitution redundancy.

Inspired by the SR rule, Rebola-Pardo [RP23] showed that the so-called *mutation logic* preserves solutions with respect to the SR rule. Through this relationship with mutation logic the modified *weak substitution redundancy (WSR)* rule was introduced by Rebola-Pardo. The key observation is that the SR rule can delete clauses, which are no longer needed for the rest of the proof, after all implications are checked. Hence, these clauses can be removed from the proof obligations, but can still be used as premises for the proof obligations. Formally, the WSR rule states that the clause C can be derived from a formula F given a subformula $G \subseteq F$ and a substitution ω if

$$F \wedge \neg C \vdash_1 (\{C\} \cup G)\upharpoonright_\omega . \tag{7}$$

The resulting formula is $G \cup \{C\}$.

Example 13. The clause x can be derived by WSR using the substitution $\omega = \{x \mapsto y\}$ from the clause $x \vee y$, which we no longer need in the rest of the proof. The negation of x propagates x to $\bot$, which makes $x \vee y$ propagate y to $\top$ resulting in the assignment $\rho_1 = \{x \mapsto \bot, y \mapsto \top\}$. The clause x is substituted to y, which is RUP, since the negation of y propagates y to $\bot$, contradicting ρ_1. This concludes the WSR check, since $x \vee y$ is deleted.

While it is possible to express advanced reasoning techniques using WSR with relatively short certificates that scale linear in the reasoning conducted by solvers, there are some limitations for this proof system, which we will discuss in Section 3.

3 Related Work

Some work related to certifying algorithms has already been mentioned in Section 2.4.1 with the certification formats for SAT like DRAT, PR, and WSR. However, the corresponding proof logging systems cannot efficiently certify all state-of-the-art reasoning techniques. The best know approach to certify parity reasoning scales cubic in the size of the formula [PR16], while our approach scales linear [GN22]. Even though it is possible to deal with simple symmetry breaking using these systems, it is not known how to certify the full range of techniques in modern symmetry breaking that our approach can certify [BGMN23].

The *DSRUP* system [TD20] has been proposed for handling symmetries in SAT solvers with a focus on solvers that want to derive symmetric versions of clauses derived by RUP with respect to known symmetries of the formula. Hence, it is only possible to derive implied constraints with this system, which makes it impossible to support pre- and inprocessing techniques. Especially, the technique of symmetry breaking is not supported, as symmetry breaking constraints are not implied, as they remove symmetric solutions.

While these systems have been designed to certify SAT solvers, they are used in ad hoc methods to certify other problem by encoding a SAT formula that proves a desired property about the problem instance and using a SAT solver to certify that this property holds. For example, this approach is used to certify solvers for hardware model checking [YBH21, FYBH24] and model counting [CCS24, BNAH23]. The main issue with these certificates is that in order to trust the certificate, we also have to trust the encoding of the property that we are interested in into a SAT formula, which can be non-trivial. In most cases this is fixed by having formally verified code with a small trust base to generate the encoding. Another difference to our approach is that the certification is decoupled from the solving. Hence, it is impossible to predict the scaling of the certificate and an error in the certificate is not linked directly to reasoning in the solver.

When it comes to the certification of MaxSAT solvers, there are other approaches that have been studied before. *MaxSAT-Resolution* [HL06] is defined for the MaxSAT formulation with soft and hard clauses. This system is extended with the redundancy notion called *inclusion redundancy* [BBL24], which allows introducing a clause C to a formula F if for a witness ω satisfying C it holds that $F{\upharpoonright}_{\neg C} \supseteq F{\upharpoonright}_{\omega}$. This rule is weaker than the redundancy notions discussed in Section 2.4.1, but automatically preserves the optimal value of the problem. The downside with MaxSAT-Resolution is that certification is only known for branch and bound algorithms and preprocessing, and it is unlikely that this system is able to certify core-guided solving [BBL24]. Furthermore, MaxSAT-Resolution has no practical relevance, as no modern MaxSAT solver that implements certification based on MaxSAT-Resolution.

There is also more straightforward extension of propagation redundancy for MaxSAT called *cost propagation redundancy*, which was proposed by Ihalainen et al. [IBJ22]. This rule is similar to the redundance-based strengthening rule in VeriPB, but additionally allows adding new variables to the objective function. This behaviour can be simulated in the VeriPB system with a redundance-based strengthening step followed by an objective update. The main downside of the work by Ihalainen et al. is that they were not able to figure out how the condition on the objective function can be checked efficiently using only clausal reasoning.

Another commonly suggested idea for MaxSAT is to check that the optimal solution satisfies all clauses and to certify optimality by running a SAT solver on the clauses together with clauses encoding that only strictly better solutions are allowed. This approach has been evaluated in Paper A, which shows that this approach has unpredictable scaling behaviour and still requires certification of the clausal encoding that only solution strictly better solutions are allowed.

To certify the correctness of mixed integer linear programming (MIP), the *VIPR* system [CGS17] was developed. This certification system is focused on LP-based branch and cut MIP solvers. Hence, the certificate format is very specialized and does not really support any other solving technique. Additionally, VIPR does not have any notion of redundancy as known for the certification of SAT solver, which makes it impossible to certify advanced presolving techniques.

There has also been recent work that used the VeriPB system to provide certification to different kinds of solvers. There are certifying constraint programming solvers using VeriPB [MM23, MMN24, FSM$^+$24, MM25] to certify reasoning with

a wide range of constraint propagators. However, it is still an open problem to provide certification for all kinds of propagators used in a modern constraint programming solver and provide formally verified encodings of constraint programming problem into a pseudo-Boolean optimization problem. For optimal classical planning, Dold et al. [DHN+25] proposed a theoretical framework that uses the VeriPB system to certify the optimality of a plan. VeriPB has also been used to certify the correctness of the Pareto front for multi objective MaxSAT solvers [JBBJ25].

4 Pseudo-Boolean Certificates

This section focuses on the certification system studied in this thesis, which constitutes one of the main contributions of this thesis. Especially, Sections 4.2.3 and 4.3 summarize the specific contributions made by this thesis towards a unified certification approach for combinatorial optimization based on pseudo-Boolean reasoning. We will first motivate the design principles guiding our certification system for combinatorial optimization. Then we will discuss our certification system in detail with a focus on the contribution of this thesis. Finally, some algorithms and data structures used in our reference implementation of a proof checker for our certification system.

4.1 Motivation

Our certification system is based on the cutting planes proof system. There are theoretical advantage of using cutting planes that are motivated by proof complexity, see Section 2.3. When comparing cutting planes to resolution, Haken [Hak85] showed exponential lower bounds in the number of steps required to refute the so-called pigeonhole principle formula, but cutting planes only requires polynomially many steps. Hence, using cutting planes can give exponentially shorter proofs for a formula than using resolution. However, it is possible to simulate extended resolution using DRAT [JHB12] and DRAT using extended resolution [KRH18]. Hence, DRAT is as strong as extended resolution as a proof system.

Our system can simulate DRAT and therefore also extended resolution, but it is not immediately clear that it could be exponentially stronger. However, as highlighted in Section 2.4, polynomial improvements in size of the certificate are important for certifying algorithms to achieve linear sized certificates. Additionally, Kołodziejczyk and Thapen [KT24] showed that the dominance-based strengthening can simulate the proof system G_1, which is above extended resolution and possibly hints towards our system being stronger than extended resolution.

Besides the theoretical advantages of cutting planes, a proof system using pseudo-Boolean constraints has the advantage that the constraints are more expressive than clauses, when comparing to SAT based certification approaches. Many problems and reasoning can be encoded more concisely. A trivial example is an at-most-one constraint stating that at most one literal of a set of n literals is true, which can be represented with one pseudo-Boolean constraint but requires n^2 many clauses. While it is possible to represent such constraints using fewer clauses, the

pseudo-Boolean constraint is still more concise and easier to grasp. It could even be discussed to lift the restrictions imposed by pseudo-Boolean constraints and more complex constraints. However, there is a trade-off between the expressiveness of the constraints and the complexity of how to handle constraints inside a checker to be sure that the checker handles them correctly.

We propose also that certification should be done in a unified multipurpose system instead of a specialized system for each component of the algorithm. For instance, image Having a certificate for preprocessing in one format and the certificate for the main solver in another format. The problem with this approach is that we need guarantees between the interplay between different certificates. This could be achieved by having a checker that can deal with both formats, but than the checker internally has to switch between different representation, e.g., if one format reasons on a graph and the other uses pseudo-Boolean constraints. This has the downside that it increases the complexity of the checker, which makes it more difficult to trust the checker. Alternatively, an interface between proofs could be defined, so that the different proofs can be checked by different checkers, which keeps each checker simple. We are actually pioneering this approach with the output section, where we can certify guarantees on a reformulated problem that can be used as input to the next checker.

Another advantage of having one unified multipurpose system is that solver authors can trust that the certification system is strong enough to certify new reasoning techniques added to the solver. Moreover, new tool, like preprocessors or symmetry breaking tools, can just be added to the solver without hassle if the tool uses the same system as the solver.

Additionally, our philosophy is that the certification should follow the reasoning of the solver as close as possible, which is contrast to just certifying the result by possibly using an independent approach. There are several advantages to this that we will discuss in the rest of this section. If the certification follows the reasoning closely, then we can get upper bounds on the size of the certificate and the additional time required to write the certificate. This is required to get efficient certifying algorithms in the sense of McConnell et al. [MMNS11].

The fact that the certificate is written while the solver is running also enables us to have certification for anytime solvers that can be interrupted and stopped arbitrarily. For instance, a solver that generates the certificate after solving would not generate any certificate if it were stopped suddenly. However, if a solver that is generating a certificate while solving is stopped, then it can finish up the certificate in the same routine that is printing the anytime result.

The closer the certificate follows the reasoning in the solver and the more detail the certificate contains, the better it can be used for detecting bugs in the reasoning of the solver. This approach can even detect bugs on instances where the solver still returns the correct result but the reasoning that led to this result is erroneous. Detecting bugs even if the returned result is correct and without even knowing what the correct result should be makes software testing and the approach of fuzzing [ZWCX22] for software testing extremely powerful. If a bug occurs, then a detailed certificate can even help to find the cause of the issue by tracing back the certificate.

A detailed certificate can also be used to deeper understand the reasoning performed by the solver. In our approach it is even possible to annotate the certificate with comments to see which parts of the certificate came from which part of the solver. This approach could even be used to extract why the solver came to this conclusion in a concise human-readable form, so that users can understand the reasons for the returned result.

All of this should make it clear that there are advantages of having a unified multipurpose certification systems that can closely follow the reasoning of the solver. Even though we have seen advantages of using pseudo-Boolean constraints to represent the reasoning, the specific format of the constraints can be debated and more general constraints than pseudo-Boolean constraints might be advantageous.

4.2 Our Pseudo-Boolean Proof System

The pseudo-Boolean proof system VeriPB was pioneered by Gocht et al. in the course of several publications [EGMN20, GMN20, GMM+20, GN22, BGMN23, Goc22]. We will first discuss some general design principles of the theoretical proof system and the representation of the proof rules in file format. Then we will review some prior work on the proof system before we will discuss the contributions to the certification system made by this thesis in detail. In this section we only focus that the system is correct and in Section 4.3 we will discuss efficient algorithms for checking the correctness of rule applications.

Our system is based on the cutting planes proof system, which means that the atoms of reasoning are pseudo-Boolean constraints. Without loss of generality we always assume that constraints are stored and treated in normalized form, since any integer linear inequality can be changed into normalized form.

The completeness of our proof system follows immediately from the completeness of the cutting planes proof system [CCT87], since any standard cutting planes proof is also a proof in our system. To show the soundness of our proof system, we use notations and definitions similar to the one used by Bogaerts et al. [BGMN23], but extend it to accommodate the new rules.

4.2.1 Proof Configuration

Before discussing the rules of our proof system, we will discuss which information the proof maintains and what the rules manipulate. A proof is a sequence of *proof configurations* $(\mathcal{C}, \mathcal{D}, f^*, \mathcal{O}_\succeq, \mathcal{S}_\succeq, \vec{z}, g, u, v, s, P^*, n)$, where

- $\mathcal{C}$ is the core set of constraint
- $\mathcal{D}$ is the derived set of constraints
- f^* is the current objective function
- $\mathcal{O}_\succeq$ is the currently active order set of constraints
- $\mathcal{S}_\succeq$ is the active auxiliary order specification set of constraints
- $\vec{z}$ is the vector of literals the active order is initialized over

- g is a Boolean variable indicating stronger guarantees on the core set
- u is the best objective value recorded
- v is the best objective value recorded while $g = \top$, i.e., the *incumbent value*
- s is a Boolean variable indicating if the strengthening-to-core mode is active
- n is the number of excluded solutions
- P^* is the current set of preserved variables.

The configuration closely resembles the global state maintained by the checker. Each rule in our proof system changes the configuration from one to another. The initial proof configuration for an optimization problem with objective f, preserved variables set P, and set of original constraints F is $(F, \emptyset, f, \emptyset, \emptyset, \emptyset, \top, \infty, \infty, \bot, P, 0)$. If we are not interested in enumerating or counting the solutions, then the set of preserved variables P is the empty set. We will explain each component in more detail and establish our desired guarantee for the configuration in the rest of this section.

The constraints known at any point in the proof are partitioned into a *core set* and a *derived set* of constraints. The core set $\mathcal{C}$ is initialized to the constraints of the input problem F. The idea of the core set is that we can have guarantees on how these constraints relate to the input constraints, e.g., both sets are equisatisfiable or have the same optimal value with respect to the objective f. All constraints added by the proof rules are added to the derived set. We do not have to guarantee anything for the constraints in the derived set, except that they are derived by a valid rule application. Järvisalo et al. [JHB12] referred to the core set as the *irredundant* set and to the derived set as the *redundant* set. The current objective function f^* is initialized to the input objective f.

Bogaerts et al. [BGMN23] introduced *orders* over assignments to make it easy to reason about symmetry breaking. The main part of an order is the pseudo-Boolean formula $\mathcal{O}_{\succeq}(\vec{u}, \vec{v}, \vec{a})$ over the variables $\vec{u}$, $\vec{v}$, and $\vec{a}$, where the length of $\vec{u}$ is equal to the length of $\vec{v}$. The formula $\mathcal{O}_{\succeq}(\vec{u}, \vec{v}, \vec{a})$ is used to compare assignments between $\vec{u}$ and $\vec{v}$ variables, i.e., $\mathcal{O}_{\succeq}(\vec{u}, \vec{v}, \vec{a})$ is satisfied by an assignment ρ if and only if $\vec{u}{\upharpoonright_\rho} \succeq \vec{v}{\upharpoonright_\rho}$. In Paper VII, we generalized orders to allow for auxiliary variables $\vec{a}$, which are defined by the auxiliary specification pseudo-Boolean formula $\mathcal{S}_{\succeq}(\vec{u}, \vec{v}, \vec{a})$. The condition for $\mathcal{S}_{\succeq}(\vec{u}, \vec{v}, \vec{a})$ is that for any partial truth value assignment ρ, that leaves all $\vec{a}$ variables unassigned, there exists an assignment extending ρ that satisfies $\mathcal{S}_{\succeq}$. The vector of literals $\vec{z}$ must have the same length as $\vec{u}$ and specifies the literals we compare assignments over.

The variable g specifies if we have stronger guarantees on the core set. For example, if g is true, then we are not allowed to turn an unsatisfiable core set satisfiable. The objective values u and v keep track of the best objective value we have encountered. The value of v can be thought of as the incumbent value or the current upper bound on the optimal value. The value of u is an upper bound on the lower bound that can be claimed at the end of the proof. The variable s indicates if the strengthening-to-core mode is activated, and we will explain this mode later in Section 4.2.3.

Recall from Section 2.2.5 that two total assignments ρ_1 and ρ_2 projected to a preserved set P are different if at least one variable in P is assigned differently in ρ_1 and ρ_2. For the variables in the current preserved set P^*, we have to guarantee that we cannot add or remove projected solutions with respect to these variables. The integer n tracks how many solution excluding constraints have been introduced, which is equivalent to the number of unique projected solutions that have been used to introduce the solution excluding constraints.

We will next define the property that we want to preserve for each proof configuration starting from the initial configuration.

Definition 1 (cf. [BGMN23, Definition 1]). For an optimization problem with objective f, preserved variables set P, and set of constraints F, a configuration $(\mathcal{C}, \mathcal{D}, f^*, \mathcal{O}_{\geq}, \mathcal{S}_{\geq}, \vec{z}, g, u, v, s, P^*, n)$ is (F, f)-*valid* if it holds that

1. For every $u' < u$, it holds that if $F \cup \{f \leq u'\}$ has m projected solutions with respect to P, then $\mathcal{C} \cup \mathcal{D} \cup \{f^* \leq u'\}$ has at least $m - n$ projected solutions with respect to P^*.
2. If $g = \top$ or $\mathcal{O}_{\geq} \neq \emptyset$, then for every total assignment ρ satisfying $\mathcal{C}$, there exists a total assignment ρ' satisfying $\mathcal{C} \cup \mathcal{D} \cup \{f^*(\rho) \geq f^*(\rho')\} \cup \mathcal{S}_{\geq}(\vec{z}{\upharpoonright_{\rho}}, \vec{z}{\upharpoonright_{\rho'}}, \vec{a}) \cup \mathcal{O}_{\geq}(\vec{z}{\upharpoonright_{\rho}}, \vec{z}{\upharpoonright_{\rho'}}, \vec{a})$ such that ρ and ρ' agree on the variables in P^*.
3. If $v < \infty$ or $n > 0$, then $F \cup \{f \leq v\}$ is satisfiable.
4. For every $v' < v$, it holds that if $g = \top$ and $\mathcal{C} \cup \{f^* \leq v'\}$ has m projected solutions with respect to P^*, then $F \cup \{f \leq v'\}$ has at least $m + n$ projected solutions with respect to P.
5. If $s = \top$, then any total assignment satisfying $\mathcal{C}$ also satisfies $\mathcal{C} \cup \mathcal{D}$.

All rules in our proof system preserve (F, f)-validity, which we will show later in the thesis for each rule, where this invariant does not follow directly. It is easy to see that the initial configuration $(F, \emptyset, f, \emptyset, \emptyset, \emptyset, \top, \infty, \infty, \bot, 0,)$ is (F, f)-valid. The following theorem combines two theorems by Bogaerts et al. and establishes the relationship between (F, f)-validity and the soundness of the proof system. The theorem has been adjusted slightly to accommodate for the changes to the proof system in this thesis compared to the system in Bogaerts et al. [BGMN23].

Theorem 1 (cf. [BGMN23, Theorem 2 and 3]). *Let F be a set of pseudo-Boolean constraints, f be a pseudo-Boolean objective, and P be a set of preserved variables. If $(\mathcal{C}, \mathcal{D}, f^*, \mathcal{O}_{\geq}, \mathcal{S}_{\geq}, \vec{z}, g, u, v, s, P^*, n)$ is an (F, f)-valid configuration, then the following holds:*

i) If $u = \infty$, $n = 0$, and $\mathcal{C} \cup \mathcal{D}$ contains the constraint $0 \geq 1$, then F is unsatisfiable.

ii) If F is unsatisfiable, then $v = \infty$ and $n = 0$.

iii) For an integer $lb \leq u$, if $n = 0$ and $\mathcal{C} \cup \mathcal{D}$ contains the constraint $f^* \geq lb$, then any solution ρ satisfying F has objective value $f(\rho) \geq lb$.

iv) If F is satisfiable, then there is a solution ρ satisfying F with objective value $f(\rho) \leq v$.

v) *If $u = \infty$ and $\mathcal{C} \cup \mathcal{D}$ contains the constraint $0 \geq 1$, then F has at most n projected solutions with respect to the preserved variables P.*

Proof. We prove the correctness of the theorem item by item. For Item i, assume for contradiction that F is satisfiable, i.e., F contains at least one solution. By Item 1 in Definition 1, the constraints in $\mathcal{C}$ are satisfiable, by choosing $u' < \infty$ large enough and because $n = 0$. Hence, this contradicts that $\mathcal{C} \cup \mathcal{D}$ contains the constraint $0 \geq 1$.

For Item ii, assume for contradiction that $v < \infty$ or $n > 0$. By Item 3 in Definition 1, $F \cup \{f \leq v\}$ is satisfiable. Hence, F itself is also satisfiable, which is a contradiction to F being unsatisfiable.

For Item iii, assume for contradiction that there is a solution ρ' satisfying F with $f(\rho') < lb$, hence ρ' is a solution satisfying $F \cup \{f \leq lb - 1\}$. Since $lb \leq u$ and $n = 0$, we have that $\mathcal{C} \cup \mathcal{D} \cup \{f^* \leq lb - 1\}$ is satisfiable by Item 1 in Definition 1. This is a contradiction to $\mathcal{C} \cup \mathcal{D}$ containing the constraint $f^* \geq lb$, as any assignment satisfying $f^* \leq lb - 1$ falsifies $f^* \geq lb$.

For Item iv, we consider two cases. If $v = \infty$, then since F is satisfiable, there is a solution satisfying F, and any solution ρ' has objective value $f(\rho') \leq \infty$. If $v < \infty$, then $F \cup \{f \leq v\}$ is satisfiable by Item 3 in Definition 1. This immediately gives us that any solution ρ' satisfying $F \cup \{f \leq v\}$ satisfies F and has an objective value $f(\rho') \leq v$.

For Item v, assume that there are $m > n$ projected solutions of F with respect to P. By Item 1 in Definition 1 and choosing $u' < \infty$ large enough, there are $m - n \geq 1$ projected solutions to $\mathcal{C} \cup \mathcal{D}$ with respect to P^*. This contradicts that $\mathcal{C} \cup \mathcal{D}$ contains the constraint $0 \geq 1$, which can not be satisfied by any assignment. □

Items 2, 4, and 5 in Definition 1 are not required to prove Theorem 1, but is required later to show that our proof rules preserve (F, f)-validity. Items i and ii in Theorem 1 are mainly relevant for decision instances, while Items iii and iv in Theorem 1 are only relevant for optimization instances.

4.2.2 Rules from Previous Work

In this section we will discuss the rules that have remained unchanged compared to Bogaerts et al. [BGMN23] and Gocht [Goc22]. If rules have been changed since the work by Bogaerts et al., we will detail the updated rules in Section 4.2.3 and omit the old version of the rule in this section. When explaining how each rule changes the configuration $(\mathcal{C}, \mathcal{D}, f^*, \mathcal{O}_{\succeq}, \mathcal{S}_{\succeq}, \vec{z}, g, u, v, s, P^*, n)$, we only explain the parts of the configuration that change and the rest of the configurations stays unchanged.

Implicational Rules The first set of rules are the rules from the *cutting planes* proof system and rules that are just syntactic sugar for cutting planes derivations. Our proof system supports *reverse unit propagations (RUP)* similar to the RUP rule in Section 2.4.1 but using pseudo-Boolean unit propagation from Section 2.2.3. Another rule is *syntactic implication*, which checks if a constraint C can be derived from another constraint D by saturation and adding literal axioms, which is

discussed in more detail in Section 4.3. A RUP step deriving C from F is just a proof by contradiction relying on unit propagation to show the implication $F \vDash C$. This is generalized by the *proof by contradiction rule* which allows deriving C from F if we give a proof deriving contradiction from $F \cup \{\neg C\}$ using only implicational rules. All of these rules add C to the derived set $\mathcal{D}$ resulting in the derived set $\mathcal{D} \cup \{C\}$, which trivially preserves (F, f)-validity, since C is implied, see Bogaerts et al. [BGMN23, Section 3.1]. We denote a derivation of the constraint C from the constraints F using only implicational rules as $F \vdash C$ and extend it to a set of constraints F' such that $F \vdash F'$ if $F \vdash D$ for all $D \in F'$.

Sanity Check Rules There are also some rules for sanity checks that do not modify the configuration, hence they are trivially sound. Their purpose is to check if the configuration is as expected, so that if a discrepancy occurs between the proof and the solver, the check immediately detects this. For a specific constraint C, we can check if a *syntactically equivalent* or *syntactically implied* constraint to C is in $\mathcal{C} \cup \mathcal{D}$ of the configuration. If this is not the case, the proof is incorrect. Since these rules do not change the configuration, they preserve (F, f)-validity.

Move to Core A constraint C can be moved from the derived set $\mathcal{D}$ to the core set $\mathcal{C}$, but not vice versa. This changes the core set from $\mathcal{C}$ to $\mathcal{C} \cup \{C\}$ and the derived set from $\mathcal{D}$ to $\mathcal{D} \setminus \{C\}$. This preserves (F, f)-validity, since the core set $\mathcal{C}$ gets more constrained but not more than $\mathcal{C} \cup \mathcal{D}$, which trivially satisfies all items in Definition 1.

Solution Logging The *solution logging* rule (or objective bound update rule in [BGMN23]) can be used to update the best recorded objective values and to derive a strict upper bounding constraint on the objective function. Given an assignment ρ that satisfies all constraints in the core set $\mathcal{C}$, we add the constraint $\{f^* \leq f^*(\rho) - 1\}$ to $\mathcal{C}$ resulting in the core set $\mathcal{C} \cup \{f^* \leq f^*(\rho) - 1\}$. The best recorded objective value u is also updated to $\min\{f^*(\rho), u\}$. If the stronger guarantee g is true, then the incumbent value v is updated to $\min\{f^*(\rho), v\}$. As shown by Bogaerts et al. [BGMN23], this rule preserves (F, f)-validity, since Items 1, 2, 4, and 5 hold trivially and Item 3 follows from Item 4.

Convenience Rules There is a rule to check if a constraint is not in database, which checks that a constraint $C \notin \mathcal{C} \cup \mathcal{D}$ and is the opposite of the syntactic equivalence sanity check. There is another rule for debugging which allows adding arbitrary constraints to the derived set. However, this rule does not preserve (F, f)-validity and the proof checker will warn the user if this rule is used.

4.2.3 Extensions to the System by This Thesis

In the included papers the proof system has been extended with the following rules, where some rules are extensions of already existing rules. For each rule it will be clarified if it is completely new or how it compares to previous versions of the rule. For all occurrences of the order, the main difference between the

order in this thesis and the order in Bogaerts et al. [BGMN23] is the addition of auxiliary variables and auxiliary specification. Again, when explaining how each rule changes the configuration $(\mathcal{C}, \mathcal{D}, f^*, \mathcal{O}_\succeq, \mathcal{S}_\succeq, \vec{z}, g, u, v, s, P^*, n)$, we only explain the parts of the configuration that change and the rest of the configurations stays unchanged.

Order Change We can *change the order* by specifying pseudo-Boolean formulas $\mathcal{O}'_\succeq(\vec{u}, \vec{v}, \vec{a})$ and $\mathcal{S}'_\succeq(\vec{u}, \vec{v}, \vec{a})$ and a list of literals $\vec{z}'$, where $\vec{u}$, $\vec{v}$, and $\vec{z}$ have the same size. This rule has been introduced in the proof system by Bogaerts et al. [BGMN23].

To ensure that the formulas $\mathcal{S}'_\succeq$ and $\mathcal{O}'_\succeq$ are encoding a valid order that is reflexive, it has to be shown that

$$\mathcal{S}'_\succeq(\vec{u}, \vec{u}, \vec{a}) \vdash \mathcal{O}'_\succeq(\vec{u}, \vec{u}, \vec{a}) \tag{8}$$

using only implicational rules. To ensure that the encoded order is transitive, it has to be shown that

$$\mathcal{S}'_\succeq(\vec{u}, \vec{v}, \vec{a}) \cup \mathcal{S}'_\succeq(\vec{v}, \vec{w}, \vec{b}) \cup \mathcal{S}'_\succeq(\vec{u}, \vec{w}, \vec{c}) \cup \mathcal{O}'_\succeq(\vec{u}, \vec{v}, \vec{a}) \cup \mathcal{O}'_\succeq(\vec{v}, \vec{w}, \vec{b}) \vdash \mathcal{O}'_\succeq(\vec{u}, \vec{w}, \vec{c}) \tag{9}$$

using only implicational rules, where $\vec{u}$, $\vec{v}$, and $\vec{w}$ have the same size and $\vec{a}$, $\vec{b}$, and $\vec{c}$ have the same size. For guaranteeing that any partial assignment not assigning any variables in $\vec{a}$ can be extended to a satisfying assignment of $\mathcal{S}'_\succeq$, the auxiliary specification $\mathcal{S}'_\succeq$ must be derivable by redundance-based strengthening witnessing only over the variables in $\vec{a}$ from an empty formula. For the soundness of this rule, it is required that the derived set $\mathcal{D}$ is empty. After all these conditions are checked, we set the active order $\mathcal{O}_\succeq$ to $\mathcal{O}'_\succeq$, the auxiliary specification $\mathcal{S}_\succeq$ to $\mathcal{S}'_\succeq$, and the order literals $\vec{z}$ to $\vec{z}'$. This rule is sound, since all items except Item 2 in Definition 1 are trivial. Item 2 holds because the derived set is empty and any satisfying assignment to $\mathcal{C}$ is also a satisfying assignment to $\mathcal{C} \cup \mathcal{D}$.

Redundance-Based Strengthening We are now discussion two rules to add constraints that are not implied. These rules behave differently when the strengthening-to-core mode $s = \top$, which is explained later in this section, so we assume $s = \bot$. *Redundance-based strengthening* [GN21] is a generalization of substitution redundancy [BT21] to pseudo-Boolean reasoning allowing arbitrary implicational derivations instead of just unit propagation. The redundance-based strengthening rule was introduced by Bogaerts et al. [BGMN23] and in Paper VII we changed the rule to handle orders with auxiliary variables. We can add a constraint C to the derived set $\mathcal{D}$ resulting in the derived set $\mathcal{D} \cup \{C\}$ by redundance-based strengthening if we are given a witness substitution ω, where $\mathrm{dom}(\omega) \cap P^* = \emptyset$, and implicational derivations showing that

$$\mathcal{C} \cup \mathcal{D} \cup \{\neg C\} \cup \mathcal{S}_\succeq(\vec{z}, \vec{z}\restriction_\omega, \vec{a}) \vdash (\mathcal{C} \cup \mathcal{D} \cup \{C\})\restriction_\omega \cup \{f^* \geq f^*\restriction_\omega\} \cup \mathcal{O}_\succeq(\vec{z}, \vec{z}\restriction_\omega, \vec{a}) . \tag{10}$$

Proposition 2. *If we derive the constraint C by redundance-based strengthening and the initial configuration $(\mathcal{C}, \mathcal{D}, f^*, \mathcal{O}_\succeq, \mathcal{S}_\succeq, \vec{z}, g, u, v, \bot, P^*, n)$ is (F, f)-valid, then the resulting configuration $(\mathcal{C}, \mathcal{D} \cup \{C\}, f^*, \mathcal{O}_\succeq, \mathcal{S}_\succeq, \vec{z}, g, u, v, \bot, P^*, n)$ is (F, f)-valid.*

Proof. We assume that the initial configuration is (F, f)-valid. Items 3 to 5 in Definition 1 follow trivially, since the initial configuration is (F, f)-valid.

For Item 1, we assume that $F \cup \{f \le u'\}$ has m projected solutions with respect to P for $u' < u$. Since the initial configuration is (F, f)-valid, $\mathcal{C} \cup \mathcal{D} \cup \{f^* \le u'\}$ has $m - n$. For every projected solution ρ^*, there exists a total assignment ρ satisfying $\mathcal{C} \cup \mathcal{D} \cup \{f^* \le u'\}$ which projects to ρ^* with respect to P^*. If C is also satisfied by ρ, then ρ^* is also a projected solution of $\mathcal{C} \cup \mathcal{D} \cup \{C\} \cup \{f^* \le u'\}$, as desired.

We will now construct a total assignment ρ' satisfying $\mathcal{C} \cup \mathcal{D} \cup \{C\} \cup \{f^* \le u'\}$ that also projects to ρ^*. If C is not satisfied by ρ, ρ satisfies $\neg C$. By (10) and since ρ can be extended to the auxiliary variables $\vec{a}$ to satisfy $\mathcal{S}_\succeq$, ρ satisfies $(\mathcal{C} \cup \mathcal{D} \cup \{C\})\restriction_\omega \cup \{f^* \ge f^*\restriction_\omega\}$. For any constraint D, it holds that if ρ satisfies $D\restriction_\omega$, then $\rho' = \rho \circ \omega$ satisfies D, since $(D\restriction_\omega)\restriction_\rho = D\restriction_{\rho\circ\omega}$. Hence, the assignment $\rho' = \rho \circ \omega$ satisfies $\mathcal{C} \cup \mathcal{D} \cup \{C\}$ and since ρ satisfies $f^* \ge f^*\restriction_\omega$, hence $f^*(\rho) \ge f^*(\rho \circ \omega) = f^*(\rho')$, ρ' also satisfies $f^* \le u'$. Since $\mathrm{dom}(\omega) \cap P^* = \emptyset$, ρ and ρ' agree on the variables in P^*, they project to the same projected solution. Hence, $\mathcal{C} \cup \mathcal{D} \cup \{C\}$ has $m - n$ projected solutions with respect to P^*.

For Item 2, we recycle the proof by Bogaerts et al. [BGMN23, Proposition 7]. We assume that $g = \top$ or $\mathcal{O}_\succeq \ne \emptyset$, as otherwise Item 2 is trivial. Consider a total assignment ρ satisfying $\mathcal{C}$. Since $g = \top$ or $\mathcal{O}_\succeq \ne \emptyset$, and Item 2 holds for the initial configuration, without loss of generality ρ also satisfies $\mathcal{C} \cup \mathcal{D}$. From the assignment ρ we construct a new assignment ρ' that satisfies $\mathcal{C} \cup \mathcal{D} \cup \{C\} \cup \{f^*(\rho) \ge f^*(\rho')\} \cup \mathcal{S}_\succeq(\vec{z}\restriction_\rho, \vec{z}\restriction_{\rho'}, \vec{a}) \cup \mathcal{O}_\succeq(\vec{z}\restriction_\rho, \vec{z}\restriction_{\rho'}, \vec{a})$.

If C is satisfied by ρ, then we can choose $\rho' = \rho$. Otherwise, let us consider $\rho' = \rho \circ \omega$, where it is important that $\mathrm{dom}(\omega) \cap P^* = \emptyset$ to guarantee that ρ and ρ' agree how the preserved variables in P^* are assigned. Since ρ is a total assignment and does not satisfy C, it satisfies $\neg C$. Hence, ρ satisfies $\mathcal{C} \cup \mathcal{D} \cup \{\neg C\}$, and it is always possible to extend ρ to the auxiliary variables $\vec{a}$ such that $\mathcal{S}_\succeq(\vec{z}\restriction_\rho, \vec{z}\restriction_{\rho'}, \vec{a})$ is satisfied by ρ. By (10), ρ also satisfies $(\mathcal{C} \cup \mathcal{D} \cup \{C\})\restriction_\omega \cup \{f^* \ge f^*\restriction_\omega\} \cup \mathcal{O}_\succeq(\vec{z}, \vec{z}\restriction_\omega)$. For a constraint D, it holds that if ρ satisfies $D\restriction_\omega$, then ρ' satisfies D, since $(D\restriction_\omega)\restriction_\rho = D\restriction_{\rho\circ\omega}$. Thus, $\mathcal{C} \cup \mathcal{D} \cup \{C\}$ is satisfied by ρ'. Similarly, we have that $\{f^*\restriction_\rho \ge f^*\restriction_{\rho'}\} \cup \mathcal{O}_\succeq(\vec{z}\restriction_\rho, \vec{z}\restriction_{\rho'})$ is satisfied for $\rho' = \rho \circ \omega$. □

Dominance-Based Strengthening *Dominance-based strengthening* [BGMN23] generalizes redundance-based strengthening even more and makes use of applying the witness iteratively to show its correctness. A constraint C can be derived by dominance-based strengthening given a witness substitution ω, where $\mathrm{dom}(\omega) \cap \omega = \emptyset$, and implicational derivations showing that

$$\mathcal{C} \cup \mathcal{D} \cup \{\neg C\} \cup \mathcal{S}_\succeq(\vec{z}, \vec{z}\restriction_\omega, \vec{a}) \vdash \mathcal{C}\restriction_\omega \cup \{f^* \ge f^*\restriction_\omega\} \cup \mathcal{O}_\succeq(\vec{z}, \vec{z}\restriction_\omega, \vec{a}) \quad (11)$$

$$\mathcal{C} \cup \mathcal{D} \cup \{\neg C\} \cup \mathcal{S}_\succeq(\vec{z}\restriction_\omega, \vec{z}, \vec{a}) \cup \mathcal{O}_\succeq(\vec{z}\restriction_\omega, \vec{z}, \vec{a}) \vdash \bot\,. \quad (12)$$

The dominance-based strengthening changes the derived set to $\mathcal{D} \cup \{C\}$.

Proposition 3. *If we derive the constraint C by redundance-based strengthening and the initial configuration $(\mathcal{C}, \mathcal{D}, f^*, \mathcal{O}_{\succeq}, \mathcal{S}_{\succeq}, \vec{z}, g, u, v, \perp, P^*, n)$ is (F, f)-valid, then the resulting configuration $(\mathcal{C}, \mathcal{D} \cup \{C\}, f^*, \mathcal{O}_{\succeq}, \mathcal{S}_{\succeq}, \vec{z}, g, u, v, \perp, P^*, n)$ is (F, f)-valid.*

Proof. We assume that the initial configuration is (F, f)-valid. Items 3 to 5 in Definition 1 follow trivially, since the initial configuration is (F, f)-valid.

For Item 1, we assume towards contradiction that Item 1 does not hold in the resulting configuration. We assume that $F \cup \{f \leq u'\}$ has m projected solutions with respect to P for $u' < u$. Hence, $\mathcal{C} \cup \mathcal{D} \cup \{C\} \cup \{f^* \leq u'\}$ has fewer than $m - n$ projected solutions with respect to P^*. Since the initial configuration is (F, f)-valid, $\mathcal{C} \cup \mathcal{D} \cup \{f^* \leq u'\}$ has at least $m - n$ projected solutions with respect to P^*. Hence, there is at least one projected solution ρ^* that no longer satisfies $\mathcal{C} \cup \mathcal{D} \cup \{f^* \leq u'\}$, but not $\mathcal{C} \cup \mathcal{D} \cup \{C\} \cup \{f^* \leq u'\}$. Let S be the set of total assignments ρ projecting to ρ^* with respect to P^*, that satisfy $\mathcal{C} \cup \mathcal{D} \cup \{f^* \leq u'\}$, but not $\mathcal{C} \cup \mathcal{D} \cup \{C\} \cup \{f^* \leq u'\}$. Let ρ be an assignment in S such that for all other assignment $\rho' \in S$ it holds that

$$\mathcal{S}_{\succeq}(\vec{z}\restriction_{\rho'}, \vec{z}\restriction_{\rho}, \vec{a}) \cup \mathcal{O}_{\succeq}(\vec{z}\restriction_{\rho'}, \vec{z}\restriction_{\rho}, \vec{a}) \tag{13}$$

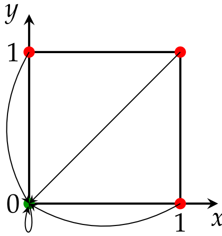
is satisfied. By construction of S, ρ does not satisfy C. Hence, ρ satisfies $C\restriction_{\omega} \cup \{f^* \geq f^*\restriction_{\omega}\} \cup \mathcal{O}_{\succeq}(\vec{z}, \vec{z}\restriction_{\omega}, \vec{a})$, so the assignment $\rho_1 = \rho \circ \omega$ satisfies C, $f^*(\rho) \geq f^*(\rho_1)$, and $\mathcal{O}_{\succeq}(\vec{z}\restriction_{\rho}, \vec{z}\restriction_{\rho_1}, \vec{a})$. Since $\mathrm{dom}(\omega) \cap P^* = \emptyset$, ρ_1 projects to ρ^*. Additionally, since ρ satisfies $\mathcal{C} \cup \mathcal{D} \cup \{\neg C\}$, ρ has to falsify $\mathcal{O}_{\succeq}(\vec{z}\restriction_{\omega}, \vec{z}, \vec{a})$ by (12), so $\mathcal{O}_{\succeq} \neq \emptyset$. Hence, we can apply Item 2 to obtain that there exists an assignment ρ_2 satisfying $\mathcal{C} \cup \mathcal{D} \cup \{f^*(\rho) \geq f^*(\rho_2)\} \cup \mathcal{S}_{\succeq}(\vec{z}\restriction_{\rho}, \vec{z}\restriction_{\rho_2}, \vec{a}) \cup \mathcal{O}_{\succeq}(\vec{z}\restriction_{\rho}, \vec{z}\restriction_{\rho_2}, \vec{a})$ that agrees with ρ_1 on the assignment to the variables in P^*. Since $f^*(\rho) \geq f^*(\rho_1) \geq f^*(\rho_2)$, ρ_2 satisfies $\mathcal{C} \cup \mathcal{D} \cup \{f^* \leq u'\}$. The assignment ρ_2 does not satisfy $\mathcal{O}_{\succeq}(\vec{z}\restriction_{\rho_2}, \vec{z}\restriction_{\rho}, \vec{a})$, as $\mathcal{O}_{\succeq}(\vec{z}\restriction_{\rho_1}, \vec{z}\restriction_{\rho_2}, \vec{a})$ is satisfied, $\mathcal{O}_{\succeq}(\vec{z}\restriction_{\rho_1}, \vec{z}\restriction_{\rho}, \vec{a})$ is not satisfied, and the order $\mathcal{O}_{\succeq}$ is transitive. Since this falsifies the condition in (13) and ρ_2 projects to ρ^*, ρ_2 is not in S and must satisfy $\mathcal{C} \cup \{C\} \cup \mathcal{D} \cup \{f^* \leq u'\}$, which is a contradiction to $\mathcal{C} \cup \{C\} \cup \mathcal{D} \cup \{f^* \leq u'\}$ being unsatisfiable.

For Item 2, we recycle the proof by Bogaerts et al. [BGMN23, Proposition 14]. We assume that $g = \top$ or $\mathcal{O}_{\succeq} \neq \emptyset$, as otherwise Item 2 is trivial. Assume for contradiction that the resulting configuration does not satisfy Item 2. Let S be the set of total assignments ρ that satisfy $\mathcal{C}$, but there is no total assignment ρ' such that $\mathcal{C} \cup \mathcal{D} \cup \{C\} \cup \{f^*(\rho) \geq f^*(\rho')\} \cup \mathcal{S}_{\succeq}(\vec{z}\restriction_{\rho}, \vec{z}\restriction_{\rho'}, \vec{a}) \cup \mathcal{O}_{\succeq}(\vec{z}\restriction_{\rho}, \vec{z}\restriction_{\rho'}, \vec{a})$ is satisfied such that ρ and ρ' agree on how to assign the variables in P^*. Since Item 2 is not satisfied, S is non-empty. Let ρ be an assignment in S such that for all $\rho' \in S$, $\rho \neq \rho'$, it holds that

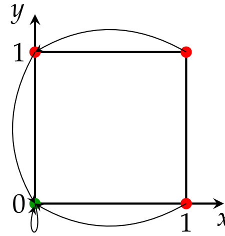
$$\{f^*(\rho') \geq f^*(\rho)\} \cup \mathcal{S}_{\succeq}(\vec{z}\restriction_{\rho'}, \vec{z}\restriction_{\rho}, \vec{a}) \cup \mathcal{O}_{\succeq}(\vec{z}\restriction_{\rho'}, \vec{z}\restriction_{\rho}, \vec{a}) \tag{14}$$

is satisfied. Since the initial configuration is (F, f)-valid, there exists a total assignment ρ_1 satisfying $\mathcal{C} \cup \mathcal{D} \cup \{f^*(\rho) \geq f^*(\rho_1)\} \cup \mathcal{S}_{\succeq}(\vec{z}\restriction_{\rho}, \vec{z}\restriction_{\rho_1}, \vec{a}) \cup \mathcal{O}_{\succeq}(\vec{z}\restriction_{\rho}, \vec{z}\restriction_{\rho_1}, \vec{a})$ such that ρ and ρ_1 agree on the assignment to the variables in P^*. Since $\rho \in S$, ρ_1 cannot satisfy C, so ρ_1 has to satisfy $\mathcal{C} \cup \mathcal{D} \cup \{\neg C\}$. By construction of $\mathcal{S}_{\succeq}$, it is possible to extend the assignment ρ_1, so that $\mathcal{S}_{\succeq}(\vec{z}\restriction, \vec{z}\restriction_{\omega}, \vec{a})$ is satisfied. Therefore, it follows from (11) that ρ_1 satisfies $C\restriction_{\omega} \cup \{f^* \geq f^*\restriction_{\omega}\} \cup \mathcal{O}_{\succeq}(\vec{z}, \vec{z}\restriction_{\omega}, \vec{a})$.

By (12) and since ρ_1 can be extended such that $\mathcal{S}_{\geq}$ is satisfied, ρ_1 does not satisfy $\mathcal{O}_{\geq}(\vec{z}\restriction_{\omega}, \vec{z}, \vec{a})$. Let ρ_2 be $\rho_1 \circ \omega$, where ρ_2 and ρ_1 agree on the assignment of the variables in P^*, since $\mathrm{dom}(\omega) \cap P^* = \emptyset$. Since ρ_1 satisfies $\mathcal{C}\restriction_{\omega}$, ρ_2 satisfies $\mathcal{C}$. It cannot be that $\rho_2 \in S$, as $\mathcal{O}_{\geq}(\vec{z}\restriction_{\rho}, \vec{z}\restriction_{\rho_1}, \vec{a})$, $\mathcal{O}_{\geq}(\vec{z}\restriction_{\rho_2}, \vec{z}\restriction_{\rho_1}, \vec{a})$, and by transitivity also $\mathcal{O}_{\geq}(\vec{z}\restriction_{\rho}, \vec{z}\restriction_{\rho_2}, \vec{a})$ are satisfied, but $\mathcal{O}_{\geq}(\vec{z}\restriction_{\rho_2}, \vec{z}\restriction_{\rho}, \vec{a})$ is not satisfied, since we show by (14) that $\mathcal{O}_{\geq}(\vec{z}\restriction_{\rho_2}, \vec{z}\restriction_{\rho_1}, \vec{a})$ is not satisfied. Since $\rho_2 \notin S$, there exists an assignment ρ' such that $\mathcal{C} \cup \mathcal{D} \cup \{C\} \cup \{f^*(\rho_2) \geq f^*(\rho')\} \cup \mathcal{S}_{\geq}(\vec{z}\restriction_{\rho_2}, \vec{z}\restriction_{\rho'}, \vec{a}) \cup \mathcal{O}_{\geq}(\vec{z}\restriction_{\rho_2}, \vec{z}\restriction_{\rho'}, \vec{a})$ is satisfied such that ρ_2 and ρ' agree on the assignment to the variables in P^*. However, by transitivity it also holds that $\{f^*(\rho) \geq f^*(\rho')\} \cup \mathcal{S}_{\geq}(\vec{z}\restriction_{\rho}, \vec{z}\restriction_{\rho'}, \vec{a}) \cup \mathcal{O}_{\geq}(\vec{z}\restriction_{\rho}, \vec{z}\restriction_{\rho'}, \vec{a})$ is satisfied, which means that ρ' cannot satisfy $\mathcal{C} \cup \mathcal{D} \cup \{C\}$. This is a contradiction, since ρ' cannot satisfy and falsify $\mathcal{C} \cup \mathcal{D} \cup \{C\}$ at the same time. □



(a) *Redundance-based strengthening with substitution $\{x \mapsto 0, y \mapsto 0\}$.*



(b) *Dominance-based strengthening with substitution $\{x \mapsto 0, y \mapsto x\}$.*

Figure 2: *Example for how the substitution patches the assignment for database $\mathcal{C} \cup \mathcal{D} = \emptyset$ and order $\mathcal{O}_{\geq}(i_1, i_2, k_1, k_2) = \{2i_1 + i_1 \geq 2k_1 + k_2\}$ loaded over the literals $\vec{z} = x, y$ to a solution satisfying also the constraint $\overline{x} + \overline{y} \geq 2$.*

To get an intuition how the substitution works for redundance- and dominance-based strengthening, we consider the example of deriving $C \doteq \overline{x} + \overline{y} \geq 2$ from an empty database $\mathcal{C} \cup \mathcal{D} = \emptyset$ under the order $\mathcal{O}_{\geq}(i_1, i_2, k_1, k_2) = \{2i_1 + i_1 \geq 2k_1 + k_2\}$ loaded over the variables $\vec{z} = x, y$. For redundance-based strengthening, the substitution has to patch any falsified assignment to an assignment also satisfying C in one step, e.g., with the substitution $\{x \mapsto 0, y \mapsto 0\}$. For dominance-based strengthening, the substitution can be applied multiple times to patch any falsified assignment to an assignment satisfying also C, e.g., with the substitution $\{x \mapsto 0, y \mapsto x\}$. This example is visualized in Figure 2.

Deletion The rules discussed so far only allow us to add constraints, but for the performance of the checker and the strength of the proof system, the system also supports the *deletion* of constraints. If we delete a constraint C from the derived set $\mathcal{D}$, we obtain the new derived set $\mathcal{D} \setminus \{C\}$ without any checks, as we have no guarantees for $\mathcal{D}$. If we delete a constraint C from the core set $\mathcal{C}$, then we can either use checked or unchecked deletion. For *checked deletion*, we have to give a substitution witness ω, where $\mathrm{dom}(\omega) \cap P^* = \emptyset$ and if $s = \top$, then the witness $\omega = \emptyset$ mapping all variables to themselves, and implicational derivations showing

$$(\mathcal{C} \setminus \{C\}) \cup \{\neg C\} \cup \mathcal{S}_{\geq}(\vec{z}, \vec{z}\restriction_{\omega}, \vec{a}) \vdash \mathcal{C}\restriction_{\omega} \cup \{f^* \geq f^*\restriction_{\omega}\} \cup \mathcal{O}_{\geq}(\vec{z}, \vec{z}\restriction_{\omega}, \vec{a}), \quad (15)$$

i.e., C can be derived for $\mathcal{C} \setminus \{C\}$ by redundance-based strengthening. Checked deletion changes the core set from $\mathcal{C}$ to $\mathcal{C} \setminus \{C\}$.

Proposition 4. *If we delete the constraint C using checked deletion from core set $\mathcal{C}$ and the initial configuration $(\mathcal{C}, \mathcal{D}, f^*, \mathcal{O}_\succeq, \mathcal{S}_\succeq, \vec{z}, g, u, v, s, P^*, n)$ is (F, f)-valid, then the resulting configuration $(\mathcal{C}, \mathcal{D} \setminus \{C\}, f^*, \mathcal{O}_\succeq, \mathcal{S}_\succeq, \vec{z}, g, u, v, s, P^*, n)$ is (F, f)-valid.*

Proof. Assume the initial configuration is (F, f)-valid. Items 1 and 3 in Definition 1 are trivially satisfied, since $\mathcal{C} \setminus \{C\} \subseteq \mathcal{C}$. For Items 2 and 4, we recycle the proof by Bogaerts et al. [BGMN23].

To show Item 2, we assume that $g = \top$ or $\mathcal{O}_\succeq \neq \emptyset$, as otherwise Item 2 is trivial. Let ρ be an assignment satisfying $\mathcal{C} \setminus \{C\}$. If ρ satisfies C, then ρ satisfies $\mathcal{C}$ and there exists a ρ' such that Item 2 holds, since the initial configuration is valid. So assume that ρ does not satisfy C. Since ρ can be extended to satisfy $\mathcal{S}_\succeq$ and by (15), ρ satisfies $\mathcal{C}\restriction_\omega \cup \{f^* \geq f^*\restriction_\omega\} \cup \mathcal{O}_\succeq(\vec{z}, \vec{z}\restriction_\omega, \vec{a})$. We obtain the assignment $\rho_1 = \rho \circ \omega$, which satisfies $\mathcal{C}$ and ρ_1 agrees with ρ on the assignment of the variables in P^*, since $\mathrm{dom}(\omega) \cap P^* = \emptyset$. Since Item 2 holds for the initial configuration, there exists a total assignment ρ' satisfying $\mathcal{C} \cup \mathcal{D} \cup \{f^*(\rho_1) \geq f^*(\rho')\} \cup \mathcal{S}_\succeq(\vec{z}\restriction_{\rho_1}, \vec{z}\restriction_{\rho'}, \vec{a}) \cup \mathcal{O}_\succeq(\vec{z}\restriction_{\rho_1}, \vec{z}\restriction_{\rho'}, \vec{a})$ and by transitivity ρ' also satisfies $\{f^*(\rho) \geq f^*(\rho')\} \cup \mathcal{S}_\succeq(\vec{z}\restriction_{\rho}, \vec{z}\restriction_{\rho'}, \vec{a}) \cup \mathcal{O}_\succeq(\vec{z}\restriction_{\rho}, \vec{z}\restriction_{\rho'}, \vec{a})$, where ρ agrees with ρ' on the assignment of the variables in P^*.

For Item 4, let ρ be a total assignment satisfying $(\mathcal{C} \setminus \{C\}) \cup \{f^* \leq v'\}$ for $v' < v$ that is projected to ρ^* with respect to P^*. If ρ satisfies C, then ρ satisfies $\mathcal{C}$ and Item 4 holds, since the initial configuration is (F, f)-valid and contains $g = \top$. Assuming that ρ does not satisfy C and since ρ can be extended such that $\mathcal{S}_\succeq$ is satisfied, ρ satisfies $\mathcal{C}\restriction_\omega \cup \{f^* \geq f^*\restriction_\omega\} \cup \mathcal{O}_\succeq(\vec{z}, \vec{z}\restriction_\omega, \vec{a})$ by (15). Hence, the assignment $\rho_1 = \rho \circ \omega$ satisfies $\mathcal{C} \cup \{f^* \leq v'\}$, since $f^*(\rho) \geq f^*(\rho_1)$, and ρ_1 also agrees with ρ on the assignment to the variables in P^*, since $\mathrm{dom}(\omega) \cap P^* = \emptyset$. Hence, $\mathcal{C} \cup \{f^* \leq v'\}$ and $(\mathcal{C} \setminus \{C\}) \cup \{f^* \leq v'\}$ have the same number of projected solutions with respect to P^*. Since Item 4 holds for the initial configuration, Item 4 also holds for the resulting configuration.

For Item 5, the argument is similar to Item 2, except that checked deletion requires the substation has to be empty $\omega = \emptyset$ if $s = \top$. Hence, if we consider a total assignment ρ that satisfies $\mathcal{C} \setminus \{C\}$, but does not satisfy C, then $\rho_1 = \rho \circ \omega$ satisfies $\mathcal{C}$ by (15). Since $\omega = \emptyset$, it holds that $\rho_1 = \rho$. Therefore, ρ satisfies $\mathcal{C} \cup \mathcal{D}$, since $s = \top$ and Item 5 holds for the initial configuration. □

For *unchecked deletion*, we only need to check that if $\mathcal{O}_\succeq \neq \emptyset$ or strengthening-to-core $s = \top$, then $\mathcal{D} = \emptyset$. By using unchecked deletion, we lose the stronger guarantees g, hence $g = \bot$ and the core set changes from $\mathcal{C}$ to $\mathcal{C} \setminus \{C\}$ in the resulting configuration.

Proposition 5. *If we delete the constraint C using unchecked deletion from core set $\mathcal{C}$ and the initial configuration $(\mathcal{C}, \mathcal{D}, f^*, \mathcal{O}_\succeq, \mathcal{S}_\succeq, \vec{z}, g, u, v, s, P^*, n)$ is (F, f)-valid, then the resulting configuration $(\mathcal{C}, \mathcal{D} \setminus \{C\}, f^*, \mathcal{O}_\succeq, \mathcal{S}_\succeq, \vec{z}, \bot, u, v, s, P^*, n)$ is (F, f)-valid.*

Proof. Assume the initial configuration is (F, f)-valid. Items 1 and 3 in Definition 1 are trivially satisfied, since $\mathcal{C} \setminus \{C\} \subseteq \mathcal{C}$. Items 2 and 4 become trivial, since g is set to false in resulting configuration. For Item 5, the derived set $\mathcal{D} = \emptyset$. Hence, $\mathcal{C} \cup \mathcal{D} = \mathcal{C}$ and Item 5 follows. □

Objective Update The main contribution of Paper IV to the proof system is the *objective update* rule that allows to change the objective. Using the objective update we change the objective from f^* to f' given implicational derivations showing that

$$\mathcal{C} \vdash \{f^* \geq f'\} \cup \{f' \geq f^*\} . \tag{16}$$

This shows that $f^* = f'$. While the objective update is still sound if $f^* \geq f'$ is derived from $\mathcal{C} \cup \mathcal{D}$, we require for simplicity that it derived by $\mathcal{C}$ only. To keep the size of the proof as small as possible, there are two ways to specify the new objective. The first way is to directly specify the updated objective f'. The second way is to specify the difference between the updated objective f' and the current objective f^*, i.e., the linear form $f' - f^*$. Proposition 6 shows that the objective update preserves (F, f)-validity.

Proposition 6. *If the initial configuration* $(\mathcal{C}, \mathcal{D}, f^*, \mathcal{O}_{\succeq}, \mathcal{S}_{\succeq}, \vec{z}, g, u, v, s, P^*, n)$ *is* (F, f)*-valid, and we use the objective update rule, then the resulting configuration* $(\mathcal{C}, \mathcal{D}, f', \mathcal{O}_{\succeq}, \mathcal{S}_{\succeq}, \vec{z}, g, u, v, s, P^*, n)$ *is also* (F, f)*-valid.*

Proof. Items 3 and 5 in Definition 1 are trivially preserved, as it not affected by the objective update. Since (16) shows that any assignment ρ satisfying $\mathcal{C}$ also satisfies $\{f^* \geq f'\} \cup \{f' \geq f^*\}$, meaning that $f^*(\rho) = f'(\rho)$. Therefore, for Item 1, any assignment ρ satisfying $\mathcal{C} \cup \mathcal{D} \cup \{f^* \leq u'\}$ for $u' < u$ also satisfies $\mathcal{C} \cup \mathcal{D} \cup \{f' \leq u'\}$. Similarly, for Item 2 let ρ be an assignment satisfying $\mathcal{C}$ and ρ' satisfy $\mathcal{C} \cup \mathcal{D} \cup \{f^*(\rho) \geq f^*(\rho')\} \cup \mathcal{S}_{\succeq}(\vec{z}\restriction_{\rho}, \vec{z}\restriction_{\rho'}, \vec{a}) \cup \mathcal{O}_{\succeq}(\vec{z}\restriction_{\rho}, \vec{z}\restriction_{\rho'}, \vec{a})$, since the initial configuration is (F, f)-valid. Since ρ and ρ' satisfy $\mathcal{C}$, it holds that $f'(\rho) = f^*(\rho) \geq f^*(\rho') = f'(\rho^prime)$, which satisfies Item 2. Finally, for Item 4, for any ρ satisfying $\mathcal{C} \cup \{f^* \leq v'\}$, ρ also satisfies $\mathcal{C} \cup \{f' \leq v'\}$, resulting in the same number of projected solutions with respect to P^*. □

Objective Equivalence Since it is now possible to change the objective, in Paper VI we introduce a rule for checking that the current objective f^* in the configuration $(\mathcal{C}, \mathcal{D}, f^*, \mathcal{O}_{\succeq}, \mathcal{S}_{\succeq}, \vec{z}, g, u, v, s, P^*, n)$ is *syntactically equivalent* to the objective specified in the rule. This rule does not change the configuration and trivially preserves (F, f)-validity.

Solution Excluding Rule In Paper VIII, we introduce a rule to enumerate projected solutions of F projected to the preserved set P. The *solution excluding* rules is similar to the solution logging rule, but can only be used when $g = \top$. Given an assignment ρ that satisfies $\mathcal{C}$ and assigns a value to each variable in P^*, then the *solution excluding constraint* $C \doteq \sum_{\{x \in P^* | \rho(x)=1\}} \overline{x} + \sum_{\{x \in P^* | \rho(x)=0\}} x \geq 1$ is added to the core set $\mathcal{C}$ resulting in the core set $\mathcal{C} \cup \{C\}$. Using this rule increments the number of excluded solutions n by 1.

Proposition 7. *If the initial configuration* $(\mathcal{C}, \mathcal{D}, f^*, \mathcal{O}_{\succeq}, \mathcal{S}_{\succeq}, \vec{z}, \top, u, v, s, P^*, n)$ *is* (F, f)*-valid,* ρ *satisfies* $\mathcal{C}$ *and assigns a value to each variable in* P^**, then the resulting configuration* $(\mathcal{C} \cup \{C\}, \mathcal{D}, f^*, \mathcal{O}_{\succeq}, \mathcal{S}_{\succeq}, \vec{z}, \top, u, v, s, P^*, n+1)$ *is* (F, f)*-valid, where the solution excluding constraint* $C \doteq \sum_{\{x \in P^* | \rho(x)=1\}} \overline{x} + \sum_{\{x \in P^* | \rho(x)=0\}} x \geq 1$.

Proof. Assume the initial configuration is (F, f)-valid. Since the solution excluding constraint is added to the core set, Items 2, 4, and 5 in Definition 1 are preserved trivially. For Item 3, it might be that $n = 0$ and $u = \infty$ in the initial configuration and changes $n = 1$ in the resulting configuration. Since, $g = \top$ and Item 4 holds, Item 3 is satisfied.

For Item 1, the solution excluding constraint C removes all solutions where the variables in P^* are mapped to the same values as in ρ. Hence, $\mathcal{C} \cup \mathcal{D} \cup \{f^* \leq u'\}$ has at most one fewer projected solution with respect to P^*, which satisfies Item 1, since n is incremented by 1. □

Preserved Set Change To support changing the current preserved set of variable P^*, Paper VIII introduces new rules to manipulate P^*. For a constraint $C \doteq \sum_i a_i \ell_i \geq A$ and a variable x, we will use for convenience the notation $x \Rightarrow C$ for the constraint $A\overline{x} + \sum_i a_i \ell_i \geq A$ and $x \Leftarrow C$ for the constraint $(\sum_i a_i - A + 1)x \sum_i a_i \overline{\ell}_i \geq \sum_i a_i - A + 1$, which can be thought of as x implies C and C implies x, respectively. To *add a variable* x to the preserved set P^*, we have to give a constraint $C \doteq \sum_i a_i \ell_i \geq A$, where all literals in ℓ_i are over variables in P^*, and implicational derivations showing that

$$\mathcal{C} \vdash \{x \Rightarrow C, x \Leftarrow C\} . \tag{17}$$

This shows that x is true if and only if C is satisfied.

Proposition 8. *Let $C \doteq \sum_i a_i \ell_i \geq A$ be a constraint, where all literals ℓ_i are over variable in P^*. If the initial configuration $(\mathcal{C}, \mathcal{D}, f^*, \mathcal{O}_{\succeq}, \mathcal{S}_{\succeq}, \vec{z}, g, u, v, s, P^*, n)$ is (F, f)-valid, then adding the variable $x \notin P^*$ to the preserved set results in the configuration $(\mathcal{C}, \mathcal{D}, f^*, \mathcal{O}_{\succeq}, \mathcal{S}_{\succeq}, \vec{z}, g, u, v, s, P^* \cup \{x\}, n)$ that is (F, f)-valid.*

Proof. Assume the initial configuration is (F, f)-valid. Since the rule only changes the preserved set P^*, Items 3 and 5 in Definition 1 are trivially satisfied. Since we add a variable to the preserved set, the number of projected solutions with respect to $P^* \cup \{x\}$ can only increase, which trivially satisfies Item 1.

For Item 2, we assume that $g = \top$ or $\mathcal{O}_{\succeq} \neq \emptyset$, as otherwise this item is trivial. Let ρ be a total assignment satisfying $\mathcal{C}$. Since the initial configuration is valid, there exists an assignment ρ' satisfying $\mathcal{C} \cup \mathcal{D} \cup \{f^*(\rho) \geq f^*(\rho')\} \cup \mathcal{S}_{\succeq}(\vec{z}\restriction_{\rho}, \vec{z}\restriction_{\rho'}, \vec{a}) \cup \mathcal{O}_{\succeq}(\vec{z}\restriction_{\rho}, \vec{z}\restriction_{\rho'}, \vec{a})$ such that ρ and ρ' agree on the assignment to the variables in P^*. Hence, ρ and ρ' project to the same solution ρ^* with respect to P^*. By (17), ρ and ρ' satisfy $\{x \Rightarrow C, x \Leftarrow C\}$, which means that x is 1 in ρ and ρ' if and only if C is satisfied by ρ^*. Hence, ρ and ρ' agree on the assignment to the variables in $P^* \cup \{x\}$.

Similarly, for Item 4, any total assignment ρ satisfying $\mathcal{C} \cup \{f^* \leq v'\}$ for $v' < v$ also satisfies $\{x \Rightarrow C, x \Leftarrow C\}$ by (17). Let ρ^* be the projected assignment of ρ with respect to P^*, hence ρ^* either satisfies or falsifies C. Since x is 1 if and only if C is satisfied by ρ^*, $\mathcal{C} \cup \{f^* \leq v'\}$ has the same number of projected solutions with respect to P^* and $P^* \cup \{x\}$. □

It is also possible to *remove a variable* x from the preserved set P^* given a constraint $C \doteq \sum_i a_i \ell_i \geq A$, where all literals in ℓ_i are over variables in $P^* \setminus \{x\}$,

and implicational derivations showing that

$$\mathcal{C} \vdash \{x \Rightarrow C, x \Leftarrow C\} \,. \tag{18}$$

Proposition 9. *Let $C \doteq \sum_i a_i \ell_i \geq A$ be a constraint, where all literals ℓ_i are over variable in $P^* \setminus \{x\}$. If the initial configuration $(\mathcal{C}, \mathcal{D}, f^*, \mathcal{O}_{\succeq}, \mathcal{S}_{\succeq}, \vec{z}, g, u, v, s, P^*, n)$ is (F, f)-valid, then removing the variable $x \in P^*$ from the preserved set results in the configuration $(\mathcal{C}, \mathcal{D}, f^*, \mathcal{O}_{\succeq}, \mathcal{S}_{\succeq}, \vec{z}, g, u, v, s, P^* \setminus \{x\}, n)$ that is (F, f)-valid.*

Proof. Assume the initial configuration is (F, f)-valid. Since the rule only changes the preserved set P^*, Items 3 and 5 in Definition 1 are trivially satisfied. Since we remove a variable from the preserved set, the number of projected solutions can only decrease, hence Items 2 and 4 is trivial.

For Item 1, we assume that $F \cup \{f^* \leq u'\}$ has m projected solutions with respect to P for $u' < u$. Since the initial configuration is valid, $\mathcal{C} \cup \mathcal{D} \cup \{f^* \leq u'\}$ has at least $m - n$ projected solutions with respect to P^*. For each projected solution ρ^* with respect to $P^* \setminus \{x\}$ satisfying $\mathcal{C} \cup \mathcal{D} \cup \{f^* \leq u'\}$, ρ^* either satisfies or falsifies C, since C only contains variables in $P^* \setminus \{x\}$. Let ρ be any total assignment satisfying $\mathcal{C} \cup \mathcal{D} \cup \{f^* \leq u'\}$ that is projected to ρ^* with respect to $P^* \setminus \{x\}$. By (17), ρ satisfies $\{x \Rightarrow C, x \Leftarrow C\}$, which means that x is 1 in ρ if and only if C is satisfied by ρ^*. Hence, any assignment projected to ρ^* agrees on the assignment of x, so for each projected solution to $\mathcal{C} \cup \mathcal{D} \cup \{f^* \leq u'\}$ with respect to $P^* \setminus \{x\}$ there is exactly one projected solution with respect to P^*. □

Strengthening-to-Core mode Paper III introduces the *strengthening-to-core mode* into the VeriPB system. This mode is disabled by default and can be turned on and off in the proof. The strengthening-to-core mode can be activated if the derived set $\mathcal{D}$ is empty, which changes the strengthening-to-core flag s in the configuration to $s = \top$. The strengthening-to-core mode can be disabled at any time, which sets $s = \bot$. The idea of this mode is that constraints derived by strengthening rules are added immediately to the core set, which guarantees that any solution satisfying $\mathcal{C}$ also satisfies $\mathcal{D}$. Enabling strengthening-to-core preserves (F, f)-validity, since Items 1 to 4 in Definition 1 are not affected. Item 5 is preserved, since $\mathcal{D} = \emptyset$, so that $\mathcal{C} = \mathcal{C} \cup \mathcal{D}$. Disabling strengthening-to-core trivially preserves (F, f)-validity.

To maintain that the proof system is sound while the strengthening-to-core is enabled, we need to be careful when deleting constraints from the core set $\mathcal{C}$. When using checked deletion, then the substitution witness ω used for (15) has to be trivial, which means that preserving (F, f)-validity of checked deletion requires an implicational derivation showing that

$$(\mathcal{C} \setminus \{C\}) \cup \{\neg C\} \vdash C \,. \tag{19}$$

This shows that any total assignment satisfying $\mathcal{C} \setminus \{C\}$ and not satisfying C satisfies C satisfies C, hence no such assignment exist and Item 5 in Definition 1 is preserved. When using unchecked deletions from the core set, then the derived set $\mathcal{D}$ must be empty, which trivially preserves (F, f)-validity. These restrictions are necessary, as it would otherwise be possible to derive contradiction for a satisfiable formula as shown by the following example.

Example 14. Let F be a satisfiable formula and we consider that the strengthening-to-core mode is enabled. For a variable y that is not used in F, we derive $y \geq 1$ using redundance-based strengthening with the witness $\{y \mapsto 1\}$, which is added to $\mathcal{C}$. Using a cutting planes derivation we can derive $y \geq 1$, which is added to the derived set. Without the restrictions, we can delete $y \geq 1$ from $\mathcal{C}$ by either checked deletion using the witness $\{y \mapsto 1\}$ or unchecked deletion. Then we derive $\overline{y} \geq 1$ by redundance-based strengthening using the witness $\{y \mapsto 0\}$. Finally, we derive the contradiction $0 \geq 1$ using cutting planes by adding $y \geq 1$, which is still in the derived set, and $\overline{y} \geq 1$.

If strengthening-to-core is enabled, then the redundance-based strengthening deriving a constraint C adds C to the core set $\mathcal{C}$ resulting in the core set $\mathcal{C} \cup \{C\}$. This requires that we are given a substitution witness ω and implicational derivations showing that

$$\mathcal{C} \cup \mathcal{D} \cup \{\neg C\} \cup \mathcal{S}_{\succeq}(\vec{z}, \vec{z}\upharpoonright_{\omega}, \vec{a}) \vdash (\mathcal{C} \cup \{C\})\upharpoonright_{\omega} \cup \{f^* \geq f^*\upharpoonright_{\omega}\} \cup \mathcal{O}_{\succeq}(\vec{z}, \vec{z}\upharpoonright_{\omega}, \vec{a}) . \tag{20}$$

The advantage of (20) compared to (10) is that we no longer have to derive $\mathcal{D}\upharpoonright_{\omega}$. Proposition 10 shows that this updated rule still preserves (F, f)-validity.

Proposition 10. *If the initial configuration $(\mathcal{C}, \mathcal{D}, f^*, \mathcal{O}_{\succeq}, \mathcal{S}_{\succeq}, \vec{z}, g, u, v, \top, P^*, n)$ is (F, f)-valid, we can derive the constraint C redundance-based strengthening resulting in the configuration $(\mathcal{C} \cup \{C\}, \mathcal{D}, f^*, \mathcal{O}_{\succeq}, \mathcal{S}_{\succeq}, \vec{z}, g, u, v, \top, P^*, n)$, which is (F, f)-valid.*

Proof. We assume that the initial configuration is (F, f)-valid. Items 2 to 5 in Definition 1 are trivially preserved, since C is added to the core set $\mathcal{C}$.

For Item 1, we modify our proof for Proposition 2. Since the initial configuration is (F, f)-valid, if $F \cup \{f \leq u'\}$ has m projected solutions with respect to P, $\mathcal{C} \cup \{f^* \leq u'\}$ has at least $m - n$ projected solutions with respect to P^*. Let ρ be a total assignment satisfying ρ satisfying $\mathcal{C} \cup \mathcal{D} \cup \{f^* \leq u'\}$ that projects to ρ^* with respect to P^*. If C is also satisfied by ρ, then $\mathcal{C} \cup \mathcal{D} \cup \{C\}\{f^* \leq u'\}$ has a solution that projects to ρ^*. If C is not satisfied by ρ, then ρ satisfies $\neg C$. By (20) and since ρ can be extended to the auxiliary variables $\vec{a}$ to satisfy $\mathcal{S}_{\succeq}$, ρ satisfies $(\mathcal{C} \cup \{C\})\upharpoonright_{\omega} \cup \{f^* \geq f^*\upharpoonright_{\omega}\}$. Hence, the assignment $\rho' = \rho \circ \omega$ satisfies $\mathcal{C} \cup \{C\}$ and since $\mathrm{dom}(\omega) \cap P^* = \emptyset$, ρ and ρ' agree on the assignment of the variables in P^*. Therefore, ρ and ρ' project to the same solution ρ^* with respect to P^* and $\mathcal{C} \cup \mathcal{D} \cup \{C\} \cup \{f^* \leq u'\}$ has the same number of projected solutions as $\mathcal{C} \cup \mathcal{D} \cup \{f^* \leq u'\}$ with respect to P^*. □

For dominance-based strengthening with strengthening-to-core, we add the derived constraint C to the core set $\mathcal{C}$ resulting in $\mathcal{C} \cup \{C\}$. This requires that we are given a substitution witness ω and implicational derivations showing (11) and (12). Since the conditions for dominance-based strengthening do not change, the proof for Proposition 3 showing that dominance-based strengthening preserves (F, f)-validity also shows that the rule preserves (F, f)-validity if strengthening-to-core is enabled. All other rules do not change when strengthening-to-core is enabled.

Proof Output In Paper V, we introduce support in the VeriPB system to state the results certified by the proof in a *proof footer*. By stating the expected result of the proof, we improve on the ad-hoc certification of optimizations problems by Bogaerts et al. [BGMN23], which could lead to the certification of incorrect results as observed by Paxian and Biere [PB26].

The first part of the footer states the *output guarantee* of the proof, which is introduced in Paper VI. This makes it possible to have a standalone certificate for problem reformulations. Let $(\mathcal{C}, \mathcal{D}, f^*, \mathcal{O}_{\succeq}, \mathcal{S}_{\succeq}, \vec{z}, g, u, v, s, P^*, n)$ be the configuration at the end of the proof. The output problem can be specified externally through an extra file or is implicitly defined as the core set $\mathcal{C}$. If the output problem is given externally with the objective f', constraints F', and preserved set P', then we check that $f' = f^*$, $P' = P^*$, and for each constraint $C' \in F'$ there is a constraint $C \in \mathcal{C}$ such that $C' \doteq C$ and vice versa.

The most basic guarantee is *derivable*, which states that the output problem can be derived from the original problem with constraints F, objective f, and preserved set P using the proof system preserving (F, f)-validity. If $g = \top$, $u = \infty$, and $n = 0$, then the stronger guarantees *equisatisfiable*, *equioptimal*, and *equienumerable* can be used, which are used for decision, optimization, and enumeration problems, respectively. The guarantee equisatisfiable guarantees that the output constraints F' are satisfiable if and only if the original constraints F are satisfiable, which follows from Items 1 and 4 in Definition 1. The guarantee equioptimal guarantees that the output problem with objective f' and constraint F' has the same optimal value as the input problem with objective f and constraint F, which also follows from Items 1 and 4 in Definition 1. The guarantee equienumerable guarantees that if F has m projected solutions with respect to P, then F' has m projected solutions with respect to P', which again follows from Items 1 and 4 in Definition 1. The equienumerable guarantee can be used for counting problem to certify that an easy to count formula has the same projected solution count as F. These guarantees can also be used if $u < \infty$ or $n > 0$ to provide the guarantees up to logged solutions.

Proof Conclusion The second part of the footer is the *conclusion* of the proof. Six types of conclusion are supported, which also includes that there is no conclusion. Let $(\mathcal{C}, \mathcal{D}, f^*, \mathcal{O}_{\succeq}, \mathcal{S}_{\succeq}, \vec{z}, g, u, v, s, P^*, n)$ be the configuration at the end of the proof. If we are just interested in deciding the satisfiability a problem with constraints F, then the conclusion can either be satisfiable or unsatisfiable. If we conclude with *unsatisfiable*, then it is checked that $u = \infty$, $n = 0$, and $\mathcal{C} \cup \mathcal{D}$ contains a contradictory constraint, which implies that the constraint $0 \geq 1$ can be trivially derived in $\mathcal{C} \cup \mathcal{D}$. Therefore, F is unsatisfiable by Item i in Theorem 1. To use the conclusion *satisfiable*, it must hold that either $v < \infty$, $n > 0$, or that a solution specified together with the conclusion satisfies F, which implies that F is satisfiable by Item ii in Theorem 1.

For optimization instances we are able to conclude with *bounds* on the optimal value of the objective function f, which can also be specified to be ∞. For the lower bound lb, it is checked that $n = 0$ and the lower bounding constraint $f \geq lb \in \mathcal{C} \cup \mathcal{D}$. Hence, lb is a lower bound on the optimal value of f subject to satisfying F by Item iii in Theorem 1. The lower bound ∞ means that F is unsatisfiable, which is verified by checking that $\mathcal{C} \cup \mathcal{D}$ contains a contradictory constraint. For the upper

bound ub, it is checked whether $v \leq ub$ or that a solution ρ specified together with the conclusion satisfies F and $f(\rho) \leq ub$. Hence, ub is an upper bound on the optimal value of f subject to satisfying F by Item iv in Theorem 1.

For enumeration problems, we can claim partial and complete enumerations. For partial enumeration, we compare the actual solution count n with the expected solution count n^* and accept if $n = n^*$, but no further checks are necessary, since the count directly follows from the definition of the configuration. A complete enumeration requires that $u = \infty$ and $\mathcal{C} \cup \mathcal{D}$ contains a contradictory constraint, which implies that we have enumerated all n projected solutions of F with respect to P by Item v in Theorem 1.

4.3 Pseudo-Boolean Proof Checking Tool

In this section, we will present the VeriPB checker and discuss some important implementation details, algorithms, and data structures that make it convenient to generate pseudo-Boolean certificates and check them efficiently. We will focus on the reference implementation VeriPB implemented in Rust.[5] The other major implementation of a checker for the VeriPB system is the formally verified proof checker CakePB implemented inside the CakeML ecosystem [TMK+19].[6] Both checkers support the whole proof system discussed in Section 4.2. However, CakePB only supports the so-called *kernel format*, which does not contain a lot of syntactic sugar in the so-called *augmented* supported by VeriPB that makes it easier to generate certificates. To bridge the gap between the two formats, VeriPB has an *elaboration* mode, where the syntactic sugar in the augmented format is translated to the kernel format, which is introduced in Paper V. We will discuss also discuss some elaboration algorithms.

4.3.1 Constraint Data Structures

VeriPB uses specialized data structures to improve the checking performance. For pseudo-Boolean constraints $\sum_i a_i \ell_i \geq A$, we need to support arbitrarily large coefficients a_i and degrees A. However, always using arbitrary integer arithmetic for all constraints comes at a cost of performance, which is why VeriPB avoids using arbitrary precision integers when possible and handles some special cases separately. The first special case is a clause $\bigvee_i \ell_i$, which is the pseudo-Boolean constraint $\sum_i \ell_i \geq 1$. Since all coefficients and the degree are one, we only store the literals ℓ_i in a vector for clauses. The second special case is a cardinality $\sum_i \ell_i \geq A$, where all coefficients are one. Hence, we store the literals ℓ_i in a vector and A as a 64-bit integer if possible. In the general case we store a constraint as a vector of pairs of coefficients and literals (a_i, ℓ_i) and the degree A. We store a_i and A as 64-bit integers if $\sum_i a_i$ and A can be represented by 64-bit integers. Otherwise, if $\sum_i a_i$ and A can be represented with 128-bit integers, we store a_i and A as 128-bit integers. If $\sum_i a_i$ or A requires more than 128-bit integers, then we use arbitrary integers for a_i and A.

[5] The source code of VeriPB is available at `https://gitlab.com/MIAOresearch/software/VeriPB`.

[6] The source code and correctness proofs of CakePB are available at `https://github.com/CakeML/cakeml/tree/master/examples/pseudo_bool`, and precompiled binaries of CakePB are available at `https://gitlab.com/MIAOresearch/software/cakepb`.

The vectors of literals or terms for the constraints are sorted by literals, where a literal is stored as an integer with the least significant bit represents if the literal is negated and the remaining bits are the variable identifier. This makes it possible to efficiently implement most cutting planes operations. However, for some operations it might be beneficial to store the terms as a hash map indexed by the literal.

4.3.2 Reverse Unit Propagation

For pseudo-Boolean unit propagation, VeriPB uses watched literal propagation [MMZ+01] with specialized data structures for different kinds of constraints as discussed by Devriendt [Dev20b]. For clauses, we use the classical two-watched literals scheme [MMZ+01]. Considering a cardinality $\sum_i \ell_i \geq A$, it is sufficient to watch $A + 1$ literals to detect any propagation. To detect any propagation for a general pseudo-Boolean constraints $\sum_i a_i \ell_i \geq A$, we have to watch enough literals such that $\sum_{\ell_i \in W} a_i \geq A + \max_i\{a_i\}$ for the set of watched literals W of the constraint. For general pseudo-Boolean constraints, we additionally maintain a lower bound on the slack resulting from the watches, the coefficients of the watches, and literals that are watched and falsified with respect to the current assignment. As usual for watched propagation, we maintain a *watch list* mapping a literal to all constraints where this literal is watched. It might be that we have to watch almost all literals in a pseudo-Boolean constraint, where it makes sense to just watch all literals and never update the watch list to improve performance [Dev20b].

To improve checking performance, it is also possible to specify propagation hints for reverse unit propagation checks similar to the LRAT format [CHH+17]. The hints are a list of constraints in $\mathcal{C} \cup \mathcal{D}$ and the negated constraint, which we propagate in the given order and the last constraint in the list is falsified by the propagated assignment. Additionally, it is also supported that only some RUP steps are annotated with hints and to make the format more flexible, which is similar to the FRAT format [BCH21].

CakePB only supports reverse unit propagation with hints, hence VeriPB needs to elaborate reverse unit propagation to add the hints. This is done by traversing the *propagation trail*, which a stack storing which assignment gets propagated by which constraint represented as a pair of literal and constraint (ℓ, C), where the top of the stack is the latest propagation. VeriPB could just print its internal propagation trail as hints, but this might contain many propagations that are unnecessary for the conflict. To extract the necessary propagations, we traverse the propagation trail starting from the conflicting constraint C and store the assignment that was required to falsify C as ρ^*. Going through the propagations, we check if a propagation (ℓ, C) was responsible for a propagation in ρ^*, then this propagation was necessary, and we add the assignments to ρ^* that were necessary to propagate ℓ. Algorithm 4 presents the described procedure.

In VeriPB, unit propagation with respect to $\mathcal{C} \cup \mathcal{D}$ is also used for solution logging to propagate partial assignments if they do not satisfy all constraints in $\mathcal{C}$. This makes it possible to only give an assignment for some variables, as the solver might not have the assignment to all variables available. For CakePB, the assignment is not propagated.

Algorithm 4: Elaboration of reverse unit propagation to get from an initial propagation trail t and conflict constraint C^* to a list of necessary constraints t^* that need to be propagated. The assignment ρ is the assignment resulting from the propagation in t.

```
1  elaborateRup(t, C*, ρ):
2  ρ* ← ∅;
3  t* ← [C*];
4  foreach ℓ ∈ lits(C*) do
5      if ρ(ℓ) = 0 then
6          ρ*(ℓ) ← 0;
7  while t ≠ ∅ do
8      (ℓ, C) ← t.pop();
9      if ρ*(ℓ) = 1 then
10         t.pushFront(C);
11         foreach ℓ' ∈ lits(C) do
12             if ρ(ℓ') = 0 then
13                 ρ*(ℓ') ← 0;
14 return t*;
```

4.3.3 Syntactic Implication

As stated in Section 4.2.2 a constraint C syntactically implies another constraint D if D can be derived from C by adding literal axioms and saturation. This description does not readily yield an efficient check for this property, but we can make some observations that will result in an efficient algorithm to check syntactic implication.

First, let us note two special cases. If C is a contradictory constraint, then D is always implied, and if D is a trivial constraint, then C always implies D. Using saturation and adding literals axioms it is impossible to increase the degree of a constraint. To decrease a coefficient, we can either apply the saturation rule if the coefficient is larger than the degree of add opposing literal axioms resulting in cancelling addition. However, the latter approach lowers the degree by the same amount as the coefficient, thus we should use cancelling division only when it is really necessary. Then we can see that one saturation step is always sufficient, since a saturation does not change the degree and lowers the coefficients more when the degree is lower. Hence, the saturation rule is used most effective if it is used when our derived constraint has the degree of D. For adding literal axioms, we have three cases. First, if the coefficient for a literal in D is larger than in C, then we do not need any cancelling additions. Second, if the coefficient for a literal in D is smaller than in C, but the degree of D is at least the coefficient in D, then also no cancelling addition in necessary. Finally, if the coefficient for a literal in D is smaller than in C and smaller than the degree of D, then we require cancelling addition with the difference of both coefficients. Therefore, the idea of Algorithm 5 to check if C syntactically implies D is to keep track of the necessary cancellations.

Algorithm 5: Check that constraint C syntactically implies constraint D.

```
1 syntacticImplication(C, D):
2 if isContradictory(C) ∨ isTrivial(D) then
3     return ⊤;
4 d ← C.degree;
5 foreach (a, ℓ) ∈ C.terms do
6     b ← coeff(C, ℓ);
7     if a > b ∧ b < D.degree then
8         d ← d − a + b;
9 return d ≥ D.degree;
```

If the degree after the necessary cancellations is at least the degree of D, then C syntactically implies D.

4.3.4 Implicational Subproofs

For implicational subproofs like (10) for redundance-based strengthening or (11), VeriPB provides some data structures and convenience features, which has the structure $F \vdash F'$. The left-hand side is referred to as the *subproof premise* and the right-hand side is the *subproof conclusion* To efficiently construct the subproof conclusion for the core and derived constraints, we maintain a mapping from literals to constraints. Hence, for a witness ω, we only need to consider constraints containing a variable in $\mathrm{dom}(\omega)$. For a specific literal ℓ in the constraint C, C is a subproof conclusion if $\omega(\ell) \neq 1$.

To avoid explicit proofs for each subproof conclusion, which can be a lot and blow up the size of the proof, VeriPB supports *autoproving* for some constraints in the subproof conclusion. Basically, VeriPB will apply some typical heuristic checks to try to derive the subproof conclusion constraints from the subproof premise. First, VeriPB checks if the subproof premise unit propagates to a conflict, which implies that all subproof conclusion constraints can be derived. Then, VeriPB will perform the following checks if the proofgoal conclusion constraint $D \in F'$ is derivable if one of the following conditions holds:

1. The constraint D is trivial.
2. The constraint D is syntactically equivalent to a constraint in F.
3. The constraint D is RUP with respect F.
4. Let ρ be the assignment unit propagated by F. There is a constraint $C \in F$ such that $C\restriction_\rho$ syntactically implies $D\restriction_\rho$.

5 Main Results of Included Papers

In this section an overview of the papers included in this thesis is given to highlight the contribution of each paper included in this thesis to the field. The contributions

in the papers are in addition to the contributions to extend the proof system as described in Section 4.2.3 and the algorithms for efficient proof checking and elaboration presented in Section 4.3. In particular, each paper studies certification for a particular combinatorial optimization paradigm or algorithm, and therefore contains details about the certification for specific types of solvers. Each paper also contains experimental results evaluating our certification approach for the given application. However, most experimental results are for older versions of VeriPB and might change with the latest version of VeriPB as discussed in Section 4.3.

There are several ways to group the included papers together. Papers I to IV do not use formal verification, hence the results obtained from the verification process do not have any formal guarantee. Papers V to VIII use the formally verified proof checker CakePB. Papers II, III, and VI are about solving MaxSAT. Papers I and IV are about solving pseudo-Boolean problems. Papers IV and VI are both about presolving and preprocessing techniques that are used in combinatorial optimization. Papers V and VIII are about solvers for graph problems.

5.1 Summary of Paper I

> Stephan Gocht, Jakob Nordström, Ruben Martins, and Andy Oertel. "Certified CNF Translations for Pseudo-Boolean Solving". Accepted for publication in *Journal of Artificial Intelligence Research*. Preliminary version in *Proceedings of the 25th International Conference on Theory and Applications of Satisfiability Testing (SAT '22)*, volume 236 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 16:1–16:25, August 2022.

Some MaxSAT and all pseudo-Boolean solvers that are based on SAT solvers encode pseudo-Boolean constraint into clauses [ES06, MML14, SN15, PRB18]. Additionally, for modelling it can be easier to state a pseudo-Boolean optimization problem and then use a tool to encode the pseudo-Boolean constraints into CNF [PS15]. While certification for the SAT solvers is well-established [Heu21], it has remained out of reach to certify encoding pseudo-Boolean constraints into CNF.

In this paper, we show how algorithms for encoding pseudo-Boolean constraints into CNF can be made certifying. We provide a general framework that can be used to certify the correctness for different encodings. The framework provides a skeleton algorithm, which only requires filling in the details for each component specific to the encoding. To illustrate how this framework can be used, we provide certifying algorithms for sequential counter [Sin05], binary adder network [ES06], totalizer [BB03], and generalized totalizer [JMM15] encodings.

By concatenating the certificate for the correctness of the encoding and a DRAT certificate [WHH14] from a SAT solver, which is syntactically modified to be compatible with VeriPB, we can get a certificate showing that the original pseudo-Boolean constraints are unsatisfiable. The certificate for satisfiability is a solution satisfying all pseudo-Boolean constraints. We further demonstrate how certifying encodings can be used to certify the correctness of the optimal value f^* obtained by MaxSAT solvers. This is done checking that the optimal solution x^* provided by the MaxSAT solver satisfies all clauses F and that $f(x^*) = f^*$. Then we

encode the pseudo-Boolean constraint $f < f^*$ that the objective function f should be strictly smaller than the optimal value f^* into clauses F^*. If the SAT solver returns unsatisfiable on the formula $F \cup F^*$, then we get a certificate showing that there is no feasible solution with a better objective value than the optimal value. However, this certificate gives no guarantee that the reasoning in the MaxSAT solver is correct for this instance.

We implemented the certification inside an encodings library and changed the SAT solver Kissat [BFF+24] to output proofs in the VeriPB format. Our experimental evaluation on the benchmark instances of the pseudo-Boolean competition 2016 [Pse16] shows that our approach can be used in practice, even though there is still room for improvements. We also evaluated the certification of optimal values returned by MaxSAT solvers and verified that the optimal values obtained in the MaxSAT Evaluation 2022 [Max22] are correct.

5.2 Summary of Paper II

Jeremias Berg, Bart Bogaerts, Jakob Nordström, Andy Oertel, and Dieter Vandesande. "Certified Core-Guided MaxSAT Solving". In *Proceedings of the 29th International Conference on Automated Deduction (CADE-29)*, volume 14132 of *Lecture Notes in Computer Science*, pages 1–22. Springer, July 2023.

Core-guided MaxSAT solving is one of the state-of-the-art approaches to solve the MaxSAT problem. In this paper for the first time we show how to add certification to core-guided MaxSAT solvers with all advanced techniques used in these solvers.

The core-guided approach is described in Section 2.2.2. The certification of the calls to the SAT solver can be used out of the box by syntactically changing the output to the VeriPB format. Since the core clause returned by the SAT solver is learnt by conflict analysis, it can be derived by a RUP step. We show that the objective reformulation in the OLL algorithm can be certified using the VeriPB system by introducing new variables through reification, which are two pseudo-Boolean constraints defining each variable. For encoding the reification constraints into CNF, we build on prior work in Paper I and in [VDB22]. To transfer a lower bound lb on the reformulated objective $f_{ref} \geq lb$ to the original objective $f_{orig} \geq lb$, we maintain the pseudo-Boolean constraint $f_{orig} \geq f_{ref}$. As core-guided algorithms are very similar, it should be straightforward to adapt our approach to any core-guided MaxSAT solver. We provide certification for advanced techniques used in core-guided solving. Specifically, we explain how core exhaustion [ABGL12], core minimization [Mar10], hardening [ABGL12], intrinsic at-most-one constraints [IMM19], lazy variable encodings [MJML14], stratification [ABGL12, MAGL11], structure sharing [IBJ21], upper bound estimation [IMM19], and weight-aware core extraction [BJ17] can be made certifying.

We implemented our certification approach into the state-of-the-art core guided MaxSAT solver CGSS [IBJ21] that uses all the aforementioned techniques. We experimentally evaluated our approach on the benchmark instances of the MaxSAT Evaluation 2022 [Max22]. The experiments revealed a bug in the reasoning of CGSS, which it inherited from its predecessor RC2 [IMM19]. This bug would not

have been discovered by just checking the optimal solution returned by the solver. This shows that our certification approach can be successfully used to detect bugs in mature tools. After fixing this bug, we were able to confirm that our approach is usable in practice. The overhead for generating the certificate while solving the instance is very low, except for some outliers due to interfacing between Python and C++. The performance for checking the certificate was sufficient, but could be improved by further engineering the VeriPB proof checker to be more efficient.

5.3 Summary of Paper III

Jeremias Berg, Bart Bogaerts, Jakob Nordström, Andy Oertel, Tobias Paxian, and Dieter Vandesand. "Certifying Without Loss of Generality Reasoning in Solution-Improving Maximum Satisfiability". In *Proceedings of the 30th International Conference on Principles and Practice of Constraint Programming (CP '24)*, volume 307 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 4:1–4:28, September 2024.

Vandesande et al. [VDB22] showed how to certify the correctness of solution-improving search (SIS) MaxSAT solvers with VeriPB by implementing certification into QMaxSAT [KZFH12], which is not the state-of-the-art for SIS MaxSAT. For instance, the modern SIS MaxSAT solver Pacose [PRB18] uses the structure of the dynamic generalized polynomial watchdog (DGPW) encoding to perform advanced without loss of generality reasoning to improve performance.

While it is possible to certify the correctness of the encoding using the framework in Paper I, the advanced reasoning using the encoding remains out of reach. To solve this issue, we introduce the idea to construct a shadow circuit over a new set of variables that mimics the circuit that is used for the DGPW encoding. Then any without loss of generality reasoning performed by the solver is certified in a shadow circuit that has the same structure except for the variables that are fixed without loss of generality. Reasoning performed on the shadow circuit is transferred back to the original circuit by redundance-based strengthening mapping each original variable to its shadow variable and the fixed variables to their fixed value.

This idea requires that the variables introduced by the original encoding only appear in encoding constraints, as this trivializes all redundance-based strengthening proof conclusions. However, the SAT solver might learn new clauses over these variables that have no shadow circuit counterpart. To mitigate this issue, we introduce the strengthening-to-core mode. With this mode enabled, all proofgoals from the clauses learnt by the SAT solver can be ignored, as they end up in the derived set of constraints. Strengthening-to-core basically enables us to use the shadow circuit approach in a complex system with many components.

Modern SIS MaxSAT solvers also employ a wide range of additional techniques to improve performance. In this paper, we provide certifying algorithms for adder caching [BBR09, PRB18], cone-of-influence encoding [PRB18], generalized boolean multilevel optimization (GBMO) [ALM09, PRB21], and TrimMaxSAT [PRB21], which are all additional techniques used in Pacose.

We also discuss in detail why the alternative certification approach in Paper I of running the MaxSAT solver without certification, checking the solution, and creating a certificate afterwards is not feasible in practice. The main reasons are:

(i) the encoding of the solution-improving constraint still need to be certified;

(ii) the running time of SAT solver and certificate size are unpredictable; and

(iii) anytime solving cannot be certified this way.

We implemented our certification approach into the MaxSAT solver Pacose. For GBMO, two different approaches are used, but during preliminary experiments we noticed that only one approach is used in practice. We only implemented certification for the used approach, but explain certification for both. Our approach is experimentally evaluated on the MaxSAT Evaluation 2023 [Max23] benchmarks. The experiments show that our approach is correct, but the performance for generating the certificate is a bit slower than what would be desirable for practical usage. We identified that some overhead in generating the certificate was due to the shadow circuits. However, some overhead seems to be due to inefficient data structures, which could be improved by further engineering effort.

5.4 Summary of Paper IV

Alexander Hoen, Andy Oertel, Ambros Gleixner, and Jakob Nordström. “Certifying MIP-Based Presolve Reductions for 0–1 Integer Linear Programs”. In *Proceedings of the 21st International Conference on the Integration of Constraint Programming, Artificial Intelligence, and Operations Research (CPAIOR '24)*, volume 14742 of *Lecture Notes in Computer Science*, pages 310–328. Springer, May 2024.

Presolving is important for the performance of MIP solving [ABG+20]. However, the VIPR system [CGS17] used for certifying MIP solving is not able to certify presolving techniques. The main issue with VIPR is that it cannot certify reasoning that removes feasible solutions as long as at least one optimal solution is preserved.

Since 0–1 ILP is a specialization of MIP, we can pioneer how presolving can be certified for 0–1 ILP to pave the way for certification of MIP presolving. This allows us to use the VeriPB system to certify the correctness of the presolving techniques, as VeriPB can reason with 0–1 ILPs and the redundance-based strengthening can deal with techniques that remove feasible solutions. We present certification for all presolving techniques applied to 0–1 ILPs by the state-of-the-art presolver PaPILO [GGH22], which covers almost all techniques implemented in PaPILO.

To certify technique that change the objective, we extend the VeriPB system with the objective update rule. Changing the objective is necessary for redundance-based strengthening, which has the proof conclusion $f \geq f\upharpoonright_{\omega}$ for an objective f and a substitution ω. Additionally, the objective update makes it possible for the certificate to closely follow the reasoning of the presolvers. We show two ways to specify the objective update. The first approach states the new objective, which is efficient when the new objective is small. The second approach states the difference between the new and old objective, which is efficient for small objective changes.

We implemented our certification approach in PaPILO and checked the proofs using VeriPB. Our approach is evaluated using MIPLIB instances converted to 0–1 ILPs [Dev20a] and the instances of the pseudo-Boolean competition 2016 [Pse16]. We experimentally verified that our approach is correct for the benchmarks and that the overhead for generating the proof is negligible. The performance for checking the certificate could be improved. A reason for the poor checking performance is that the presolver writes preconstructed proof artefacts to the certificate, whereas the checker actually has to check the correctness of these steps. For techniques relying on propagation, we compare RUP against cutting planes certification concluding that RUP should be preferred due to smaller certificates.

5.5 Summary of Paper V

Stephan Gocht, Ciaran McCreesh, Magnus O. Myreen, Jakob Nordström, Andy Oertel, and Yong Kiam Tan. “End-to-End Verification for Subgraph Solving”. In *Proceedings of the 38th AAAI Conference on Artificial Intelligence (AAAI '24)*, pages 8038–8047, Febuary 2024.

Certifying algorithms move from trusting the solver to trusting the checker. For complex proof systems like DRAT or VeriPB, it can be difficult to trust the checker. The common way to address this issue is with a formally verified checker. For SAT solving certificates, there are the checkers cake_lpr [THM23] and gratchk [Lam20]. This paper introduces CakePB, the first formally verified checker for the VeriPB system and verified in the CakeML ecosystem [MO14, GMKN17]. Using the CakeML ecosystem, we only have to trust are a few components that are either easy to check or extensively validated. This gives us the highest assurance standard for binary code extraction [KMTM18].

To assure that the certificate is certifying the correct problem, the formally verified checker takes the original problem given to the solver as input and not just a pseudo-Boolean encoding of it. Thus, if the checker accepts the certificate, we can be sure that the solver output is correct with respect to the input problem. However, we still maintain that the checker is flexible and easily extensible to different problems. This is achieved by separating the checker in a frontend, which encodes the original problem into a pseudo-Boolean optimization problem, and a backend, which performs reasoning based on the pseudo-Boolean encoding. The final conclusion obtained at the end of the proof, e.g., unsatisfiable, is also translated back by the frontend to the original problem domain.

Due to the poor performance of formally verified unit propagation, we elaborate the proof before it is checked by CakePB [CHH+17], which is described in Section 4.3. In elaboration VeriPB adds more details to the proofs, where RUP steps are elaborated in this paper to cutting planes derivations. We even show that elaboration can be used to synchronize slightly different encodings used in the solver and the formally verified checker. To demonstrate that our approach works in practice, we implemented formally verified checkers for the problems of subgraph isomorphism, clique, and maximum common (connected) induced subgraph supporting all rules in the VeriPB system. For the purpose of this thesis,

the details for the specific graph problems are omitted. A detailed description of the problems can be found in the respective papers [GMN20, GMM+20]. All checkers use the same backend. The certifying algorithm to solve the graph problems [GMN20, GMM+20] are slightly modified to synchronize the encodings. Additionally, checked deletion [BGMN23] has been fully implemented into VeriPB.

We conducted experiments on the same benchmark used in [GMN20, GMM+20]. We experimentally validated that our approach and the implementation are correct for the benchmarks. The running time to check and elaborate the proof is slightly larger than just checking the proof. Checking the elaborated proof with CakePB is faster than with VeriPB, as the elaborated proof contains extra details.

5.6 Summary of Paper VI

Hannes Ihalainen, Andy Oertel, Yong Kiam Tan, Jeremias Berg, Matti Järvisalo, Magnus O. Myreen, and Jakob Nordström. "Certified MaxSAT Preprocessing". In *Proceedings of the 12th International Joint Conference on Automated Reasoning (IJCAR '24)*, volume 14739 of *Lecture Notes in Computer Science*, pages 396–418. Springer, July 2024.

Preprocessing is an important technique for modern MaxSAT solving [KBSJ17, IBJ22]. We present certifying algorithms using the VeriPB system for all preprocessing techniques in the MaxSAT preprocessor MaxPre [IBJ22]. By using VeriPB we have the advantage that it is possible to integrate certification for additional techniques, like advanced symmetry breaking.

To certify standalone problem reformulation tools, like preprocessors, we extended the VeriPB system with an output section. We certify that the problem resulting from the core constraints together with the objective at the end of the proof has the same optimal value as the original problem. Additionally, the checker verifies that the core constraints and the objective at the end of the proof match the reformulated problem returned by a preprocessor. We extended the formally verified checker CakePB with support for the output section and added a frontend for MaxSAT problems. For MaxSAT preprocessing certificates, this means that we get formal guarantee that the original problem given in MaxSAT format has the same optimal value as the reformulated problem in MaxSAT format.

We implemented certification into the MaxSAT preprocessor MaxPre. The correctness of our approach is experimentally verified using the benchmarks from the MaxSAT Evaluation 2023 [Max23]. The overhead for generating the certificate is slower than desired, but a lot of time is spent on renaming variables required to match the MaxSAT format, which could be improved by adding a dedicated rule. Most of the time for checking the proof is spent in VeriPB. However, Figure 3 shows how the checking performance of VeriPB has been improved since the paper was written and Figure 4 compares the performance of VeriPB to MaxPre.

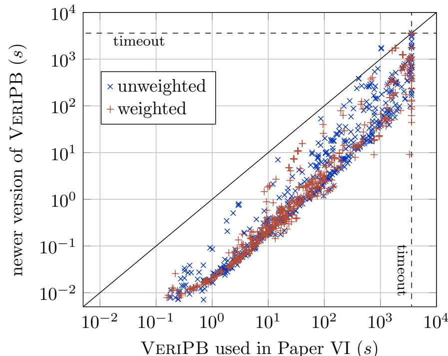


Figure 3: *VeriPB version used in Paper VI vs. latest VeriPB version.*

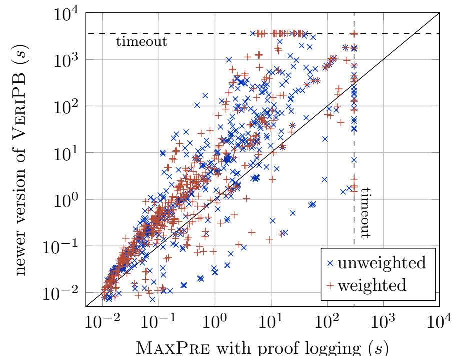


Figure 4: *Checking performance of latest VeriPB version for experiments in Paper VI.*

5.7 Summary of Paper VII

Markus Anders, Bart Bogaerts, Benjamin Bogø, Arthur Gontier, Wietze Koops, Ciaran McCreesh, Magnus O. Myreen, Jakob Nordström, Andy Oertel, Adrian Rebola-Pardo, and Yong Kiam Tan. "Faster Certified Symmetry Breaking Using Orders With Auxiliary Variables". In *Proceedings of the 40th Annual AAAI Conference on Artificial Intelligence (AAAI '26)*, January 2026.

Symmetry breaking is an important preprocessing technique for SAT [Sak21]. There has been prior work on certifying symmetry breaking using the VeriPB system by Bogaerts et al. [BGMN23]. However, due to recent advancements in symmetry breaking [ABR24] it has become clear that the certification approach by Bogaerts et al. can not keep up with the performance of state-of-the-art symmetry breaking. Specifically, the certificate size scales quadratically in the running time of the symmetry breaker.

We modify the proof system by Bogaerts et al. to achieve linear scaling for certification of symmetry breaking. This is achieved by allowing auxiliary variables to be used for the definition of the order as described in Section 4.2.1. The auxiliary variables are then used to encode the order in linear size. The new definition of the order makes it possible to generate the certificate for breaking a symmetry in linear size. Furthermore, the more concise encoding of the variables makes it possible to check the certificate more efficiently. We extend the proof format to efficiently handle orders with auxiliary variables. Support for the changed rule is implemented in VeriPB and the formally verified proof checker CakePB.

To experimentally evaluate our approach, we implemented certification with our approach and the approach by Bogaerts et al. into the state-of-the-art SAT symmetry breaking tool satsuma [ABR24]. For the benchmark instances we were able to confirm the correctness of our certification approach. We evaluate our implementation on crafted benchmarks, which shows that our approach only adds a moderate overhead for generating the certificate, while the approach by Bogaerts et al. significantly slows down satsuma. Additionally, the running time of VeriPB

and CakePB to check the certificates has improved significantly. We also evaluate on all instances from that SAT competition 2020 to 2024 where satsuma broke symmetries. These experiments show again a small overhead for satsuma. While these experiments show a big performance improvement for checking, there still remains room to further improve the performance.

5.8 Summary of Paper VIII

Ciaran McCreesh, Jakob Nordström, Andy Oertel, and Yong Kiam Tan. "Proof Logging for Projected Enumeration (and Counting?) Problems in VeriPB". *in review*.

We study (projected) enumeration and counting problems as described in Section 2.2.5. There has been prior work on certifying counting problems [FHR22, BNAH23, CCS24] and even approximate model counting [TYS+24], but for (projected) enumeration and projected counting problems no certification approach is known. This paper shows how certification for all of these problems can be integrated into the VeriPB system with more advanced rules, like redundance- and dominance-based strengthening.

To support the certification of (projected) enumeration problems, we introduced the notion of preserved sets into the proof system, which is the set of variables we are interested in. In the case of non-projected enumeration, this can be thought of as the set of all variables in the original problem. For a preserved set P, a solution is recorded by giving a satisfying assignment ρ and introducing the solution excluding constraint $\sum_{\{x \in P | \rho(x)=1\}} \overline{x} + \sum_{\{x \in P | \rho(x)=0\}} x \geq 1$ to the core set, which says that at least one preserved variable has to be assigned differently. A complete enumeration is certified by deriving a contradiction, which shows that no further solutions exists. To guarantee that deletion does add new solutions, checked deletion has to be used, except for establishing the contradiction. To ensure that redundance- or dominance-based strengthening does not remove solutions, the domain of the witnesses is not allowed to contain variables in the preserved set.

Our proposed certification approach for (projected) counting problems, is to prove that the original formula has the same number of (projected) solutions as a formula where counting the solutions is trivial. This is similar to the CPOG framework [BNAH23], but we suggest a more generic approach. Since the resulting formula might be over a different set of variables, we introduce rules that make it possible to change the preserved set, as long as the solution count is preserved. At the end of the proof, we establish that the original formula and the current core set have the same (projected) count.

We have implemented the support for (projected) enumeration certificates and for proving an equivalent number of counts into VeriPB and CakePB, and also implemented enumeration frontends for CakePB. We modified the existing certifying enumerations solvers [GMM+20, GMN22] to work with our new framework. Our experiments show the correctness of our approach on the test instances. For counting, it remains as future work to implement the certification presented in this paper into a solver, but we show that this should be possible in principle.

5.9 Further Contributions

First, we consider the additional papers co-authored by Andy Oertel which are not included in this thesis, which further show that the VeriPB system can be used for different solvers. We show how to do certification for solvers that reason with states and transitions between them, which can be found in dynamic programming solvers and solvers based on decision diagrams [DMM+24]. We demonstrate how the VeriPB system can be used to efficiently obtain end-to-end certification and checking for pseudo-Boolean solvers [KLBM+25] and state-of-the-art graph colouring solvers including improved certification for the Mycielski bound [DKK+26]. Finally, we developed a proof trimmer for the VeriPB system including the advanced redundance- and dominance-based strengthening rules [ABB+26]. The idea of trimming is to remove all proof steps that are not necessary for deriving the conclusion at the end of the proof.

With respect to VeriPB, there are further minor contributions that have not ended up in any peer-reviewed publications. First and foremost, VeriPB can be used as a checker in the SAT competition and the pseudo-Boolean competition since 2023, which requires a toolchain that checks the proof against a formally verified checker. For the SAT competition 2023 [BHI+23], we pioneered the formal verification toolchain for decision instances, which was later published in Paper V. To improve the performance of the toolchain for the SAT competition 2024 [HIJS24], we added propagation hints for RUP checks similar to LRAT [CHH+17]. For the pseudo-Boolean competition 2024 [Pse24], elaboration for the advanced autoproving technique of substituted database implication was added.

To avoid the synchronization of encodings between the solver and CakePB and to simplify the certification for classical optimal planning algorithms [DHN+25], VeriPB can handle labels to identify constraints, which are removed by elaboration.

6 Conclusions and Future Work

This thesis shows how pseudo-Boolean reasoning can be used to get efficient certifying algorithms for different combinatorial optimization paradigms. Most notably, we demonstrate how to certify state-of-the-art MaxSAT algorithms that use solution-improving and core-guided search in Papers I to III. Specifically, Paper I introduces a general approach to certify CNF encodings of pseudo-Boolean constraints used everywhere in MaxSAT solving to handle the pseudo-Boolean objective function.

We also present certification for state-of-the-art preprocessing (aka. presolving) techniques, which are crucial for the performance modern combinatorial optimization solvers. The certification approach is demonstrated for all MIP presolving techniques that preserve the variable domain $\{0, 1\}$ (see Paper IV) and for all techniques used in MaxSAT preprocessing (see Paper VI).

To guarantee that certificates generated in our VeriPB format can be trusted, we developed a formally verified proof checker in Papers V and VI that has full support for our proof system. With the approach of formal verification, the amount of code that has to be trusted is minimized and parts of the code are audited independently. However, the performance of formally verified software can not

compete with untrusted software, which makes it impossible to implement some syntactic sugar rules that the untrusted checker supports. To bridge this gap, the untrusted checker has been extended with elaboration to translate the syntactic sugar to other rules that the formally verified checker supports.

6.1 Short Term Future Work

This thesis introduction is concluded with a discussion of future work on certifying combinatorial optimization based on pseudo-Boolean reasoning. We start with some short term future work, which should be immediately in reach.

First and foremost, even though the performance of VeriPB and CakePB has improved significantly, there are some ideas to further improve the performance of both tools. Since formally verified results are important, the most important metric is the total running time of VeriPB in elaboration mode and CakePB checking the kernel proof. Specifically, the elaboration algorithm to generate the hints for reverse unit propagation is currently very naive, which makes VeriPB fast, but fewer hints might be sufficient, which might improve the performance of CakePB.

To demonstrate that our proof logging approach is general, it should be possible certify incremental solving. Incremental solvers use information derived from previous calls to the solving engine to speed up subsequent calls, where it is not trivial which derivations can be reused. While there has been previous work on certifying incremental SAT solvers [FPFB24], incremental MaxSAT and pseudo-Boolean solvers are getting popular and are not certifying.

While Paper VIII demonstrates in principle how (projected) counting solvers can be certified, it should be possible to implement the certification in a solver. Furthermore, by defining a specific format for trivially countable formulas, the VeriPB system could provide end-to-end verification for solution counts by analysing the constraints at the end of the proof.

6.2 Long Term Future Work

Finally, we discuss some long term research directions for certifying algorithms. The current proof system operates with pseudo-Boolean constraint, but it should be possible to generalize the rules to handle constraints with rational coefficients and integer or even rational variables. There exists preliminary work on such an extension of the VeriPB system [DEGH23], but it is still unclear how efficient logging and checking can be implemented in practice.

While parallel and distributed combinatorial solvers are becoming more and more mainstream [SRB25], checkers are currently completely sequential. There are several ideas to make parallel or distributed proof checking work. For distributed checking the proof could be divided into consecutive parts and each part is checked independently, which requires knowing the database at the start of each part. Another idea that is more viable in concurrent settings is checking the correctness of each rule in parallel, which comes with challenges in maintaining the database.

As proof checking becomes comparable to the performance of solvers [PFB23, Lam24], it becomes viable to run a proof checker in parallel to a solver. This would reduce the time to receive a verified result for a problem and the solver immediately

fails when its reasoning is incorrect. To reduce the overhead for reading and writing the proof, the data structure for each rule could directly be constructed by the solver and then send to the checker through a common interface.

To make it as easy as possible to implement certification in more and more solvers, a library that implements certification for common reasoning techniques might be helpful, as we observed that similar reasoning is used in all kinds of solvers. Alternatively, the proof format could be made extensible, so that new rules can be defined in terms of existing rules, which makes it possible to replace these new rules with standard VeriPB rules in elaboration.

Currently, VeriPB only has one fixed mode for how strict the reasoning in a proof is checked, but different applications might require different modes. If we are only interested in knowing that the result is correct, we could use a permissive mode that tries to patch a proof if the reasoning is slightly wrong. The other extreme is a strict mode where VeriPB enforces that all reasoning in the proof is correct without any ambiguities, which can be helpful for discovering bugs early.

The permissive mode and the autoproving performed by VeriPB in general could be made even more powerful by integrating a pseudo-Boolean solver, e.g., RoundingSat. The idea would be to call the solver on the formula and the negation of the constraint that we want to derive with the goal of deriving contradiction. This can be compared to Sledgehammer [BN10] for the higher-order logic proof assistant Isabelle [NWP02]. However, using a solver to check if constraints can be derived breaks the guarantee that proofs are checkable in polynomial time.

For developing and prototyping proofs it might be beneficial to have an interactive mode in the checker. For example, the checker runs until a specified point in the proof and then the user can interact with the checker using the proof rules. Similar to a debugger the user might also be able to explore the current state of the checker by viewing the constraints available at the given point in the proof.

Finally, proofs can be analysed to better understand the reasoning performed by solvers, i.e., which techniques and heuristics are making the most progress towards the result. This might also unveil performance bugs due to suboptimal reasoning, e.g., a contradictory constraint was derived early in the proof without the solver noticing. The analysis can also be used to explain why the solver came to its result in a format that is understandable to humans.

References

[AB16] Sanjeev Arora and Boaz Barak. *Computational Complexity: A Modern Approach*. Cambridge University Press, New York, 4th edition, 2016.

[ABB+26] Berhan Oumer Adame, Bart Bogaerts, Benjamin Bogø, Simon Dold, Arthur Gontier, Wietze Koops, Ciaran McCreesh, Matthew McIlree, Jakob Nordström, Andy Oertel, Adrian Rebola-Pardo, and Mark Turnbull. Veripb proof trimming. 2026. manuscript in progress.

[ABG+20] Tobias Achterberg, Robert E. Bixby, Zonghao Gu, Edward Rothberg, and Dieter Weninger. Presolve reductions in mixed integer programming. *INFORMS Journal on Computing*, 32(2):473–506, 2020.

[ABGL12] Carlos Ansótegui, María Luisa Bonet, Joel Gabàs, and Jordi Levy. Improving SAT-based weighted MaxSAT solvers. In *Proceedings of the 18th International Conference on Principles and Practice of Constraint Programming (CP '12)*, volume 7514 of *Lecture Notes in Computer Science*, pages 86–101. Springer, October 2012.

[ABR24] Markus Anders, Sofia Brenner, and Gaurav Rattan. Satsuma: Structure-based symmetry breaking in SAT. In *Proceedings of the 27th International Conference on Theory and Applications of Satisfiability Testing (SAT '24)*, volume 305 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 4:1–4:23, August 2024.

[ABS13] Cyrille Artho, Armin Biere, and Martina Seidl. Model-based testing for verification back-ends. In Margus Veanes and Luca Viganò, editors, *Tests and Proofs*, pages 39–55, Berlin, Heidelberg, 2013. Springer Berlin Heidelberg.

[Ach07] Tobias Achterberg. *Constraint Integer Programming*. PhD thesis, Technische Universität Berlin, 2007. Available at `https://opus4.kobv.de/opus4-zib/files/1112/Achterberg_Constraint_Integer_Programming.pdf`.

[ACMS15] Rehan Abdul Aziz, Geoffrey Chu, Christian Muise, and Peter Stuckey. #∃SAT: Projected model counting. In Marijn Heule and Sean Weaver, editors, *Theory and Applications of Satisfiability Testing – SAT 2015*, pages 121–137, Cham, 2015. Springer International Publishing.

[AGJ+18] Özgür Akgün, Ian P. Gent, Christopher Jefferson, Ian Miguel, and Peter Nightingale. Metamorphic testing of constraint solvers. In *Proceedings of the 24th International Conference on Principles and Practice of Constraint Programming (CP '18)*, volume 11008 of *Lecture Notes in Computer Science*, pages 727–736. Springer, August 2018.

[AH14] André Abramé and Djamal Habet. ahmaxsat: Description and evaluation of a branch and bound max-sat solver. *Journal on Satisfiability, Boolean Modeling and Computation*, 9(1):89–128, 2014.

[AKMS12] Benjamin Andres, Benjamin Kaufmann, Oliver Matheis, and Torsten Schaub. Unsatisfiability-based optimization in clasp. In *Technical Communications of the 28th International Conference on Logic Programming (ICLP '12)*, volume 17 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 211–221, September 2012.

[ALM09] Josep Argelich, Inês Lynce, and João P. Marques-Silva. On solving Boolean multilevel optimization problems. In *Proceedings of the 21st International Joint Conference on Artificial Intelligence (IJCAI '09)*, pages 393–398, July 2009.

[AV07] T Andersson and P Värbrand. Decision support tools for ambulance dispatch and relocation. *Journal of the Operational Research Society*, 58(2):195–201, 2007.

[AW13] Tobias Achterberg and Roland Wunderling. Mixed integer programming: Analyzing 12 years of progress. In Michael Jünger and Gerhard Reinelt, editors, *Facets of Combinatorial Optimization*, pages 449–481. Springer, 2013.

[Bat68] Kenneth E. Batcher. Sorting networks and their applications. In *Proceedings of the Spring Joint Computer Conference of the American Federation of Information Processing Societies (AFIPS '68)*, volume 32, pages 307–314, April 1968.

[BB03] Olivier Bailleux and Yacine Boufkhad. Efficient CNF encoding of Boolean cardinality constraints. In *Proceedings of the 9th International Conference on Principles and Practice of Constraint Programming (CP '03)*, volume 2833 of *Lecture Notes in Computer Science*, pages 108–122. Springer, September 2003.

[BBL24] Ilario Bonacina, Maria Luisa Bonet, and Massimo Lauria. MaxSAT Resolution with Inclusion Redundancy. In Supratik Chakraborty and Jie-Hong Roland Jiang, editors, *27th International Conference on Theory and Applications of Satisfiability Testing (SAT 2024)*, volume 305 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 7:1–7:15, Dagstuhl, Germany, 2024. Schloss Dagstuhl – Leibniz-Zentrum für Informatik.

[BBN+23] Jeremias Berg, Bart Bogaerts, Jakob Nordström, Andy Oertel, and Dieter Vandesande. Certified core-guided MaxSAT solving. In *Proceedings of the 29th International Conference on Automated Deduction (CADE-29)*, volume 14132 of *Lecture Notes in Computer Science*, pages 1–22. Springer, July 2023.

[BBR09] Olivier Bailleux, Yacine Boufkhad, and Olivier Roussel. New encodings of pseudo-Boolean constraints into CNF. In *Proceedings of the 12th International Conference on Theory and Applications of Satisfiability Testing (SAT '09)*, volume 5584 of *Lecture Notes in Computer Science*, pages 181–194. Springer, June 2009.

[BBVC13] Felix Brandt, Reinhard Bauer, Markus Völker, and Andreas Cardeneo. A constraint programming-based approach to a large-scale energy management problem with varied constraints: A solution approach to the roadef/euro challenge 2010. *Journal of Scheduling*, 16(6):629–648, 2013.

[BCCZ99] Armin Biere, Alessandro Cimatti, Edmund Clarke, and Yunshan Zhu. Symbolic model checking without bdds. In W. Rance Cleaveland, editor, *Tools and Algorithms for the Construction and Analysis of Systems*, pages 193–207, Berlin, Heidelberg, 1999. Springer Berlin Heidelberg.

[BCH21] Seulkee Baek, Mario Carneiro, and Marijn J. H. Heule. A flexible proof format for SAT solver-elaborator communication. In *Proceedings of the 27th International Conference on Tools and Algorithms for the Construction*

and Analysis of Systems (TACAS '21), volume 12651 of *Lecture Notes in Computer Science*, pages 59–75. Springer, March-April 2021.

[BFF+24] Armin Biere, Tobias Faller, Katalin Fazekas, Mathias Fleury, Nils Froleyks, and Florian Pollitt. CaDiCaL, Gimsatul, IsaSAT and Kissat entering the SAT Competition 2024. In Marijn Heule, Markus Iser, Matti Järvisalo, and Martin Suda, editors, *Proc. of SAT Competition 2024 – Solver, Benchmark and Proof Checker Descriptions*, volume B-2024-1 of *Department of Computer Science Report Series B*, pages 8–10. University of Helsinki, 2024.

[BFT11] Frédéric Besson, Pascal Fontaine, and Laurent Théry. A Flexible Proof Format for SMT: a Proposal. In Pascal Fontaine and Aaron Stump, editors, *First International Workshop on Proof eXchange for Theorem Proving - PxTP 2011*, Wroclaw, Poland, August 2011.

[BGMN23] Bart Bogaerts, Stephan Gocht, Ciaran McCreesh, and Jakob Nordström. Certified dominance and symmetry breaking for combinatorial optimisation. *Journal of Artificial Intelligence Research*, 77:1539–1589, August 2023. Preliminary version in *AAAI '22*.

[BHI+23] Tomas Balyo, Marijn Heule, Markus Iser, Matti Järvisalo, and Martin Suda, editors. *Proceedings of SAT Competition 2023: Solver, Benchmark and Proof Checker Descriptions*. Department of Computer Science Series of Publications B. Department of Computer Science, University of Helsinki, Finland, 2023.

[BHvMW21] Armin Biere, Marijn J. H. Heule, Hans van Maaren, and Toby Walsh, editors. *Handbook of Satisfiability*, volume 336 of *Frontiers in Artificial Intelligence and Applications*. IOS Press, 2nd edition, February 2021.

[BJ17] Jeremias Berg and Matti Järvisalo. Weight-aware core extraction in SAT-based MaxSAT solving. In *Proceedings of the 23rd International Conference on Principles and Practice of Constraint Programming (CP '17)*, volume 10416 of *Lecture Notes in Computer Science*, pages 652–670. Springer, August 2017.

[BJK21] Armin Biere, Matti Järvisalo, and Benjamin Kiesl. Preprocessing in SAT solving. In Biere et al. [BHvMW21], chapter 9, pages 391–435.

[BJM21] Fahiem Bacchus, Matti Järvisalo, and Ruben Martins. Maximum satisfiabiliy. In Biere et al. [BHvMW21], chapter 24, pages 929–991.

[Bla37] Archie Blake. *Canonical Expressions in Boolean Algebra*. PhD thesis, University of Chicago, 1937.

[BLB10] Robert Brummayer, Florian Lonsing, and Armin Biere. Automated testing and debugging of SAT and QBF solvers. In *Proceedings of the 13th International Conference on Theory and Applications of Satisfiability Testing (SAT '10)*, volume 6175 of *Lecture Notes in Computer Science*, pages 44–57. Springer, July 2010.

[BN10] Sascha Böhme and Tobias Nipkow. Sledgehammer: Judgement day. In Jürgen Giesl and Reiner Hähnle, editors, *Automated Reasoning*, pages 107–121, Berlin, Heidelberg, 2010. Springer Berlin Heidelberg.

[BN21] Samuel R. Buss and Jakob Nordström. Proof complexity and SAT solving. In Biere et al. [BHvMW21], chapter 7, pages 233–350.

[BNAH23] Randal E. Bryant, Wojciech Nawrocki, Jeremy Avigad, and Marijn J. H. Heule. Certified knowledge compilation with application to verified model counting. In *Proceedings of the 26th International Conference on Theory and Applications of Satisfiability Testing (SAT '23)*, volume 271 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 6:1–6:20, July 2023.

[BT21] Sam Buss and Neil Thapen. DRAT and propagation redundancy proofs without new variables. *Logical Methods in Computer Science*, 17(2):12:1–12:31, April 2021. Preliminary version in *SAT '19*.

[CCS24] Sravanthi Chede, Leroy Chew, and Anil Shukla. Circuits, Proofs and Propositional Model Counting. In Siddharth Barman and Sławomir Lasota, editors, *44th IARCS Annual Conference on Foundations of Software Technology and Theoretical Computer Science (FSTTCS 2024)*, volume 323 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 18:1–18:23, Dagstuhl, Germany, 2024. Schloss Dagstuhl – Leibniz-Zentrum für Informatik.

[CCT87] William Cook, Collette Rene Coullard, and György Turán. On the complexity of cutting-plane proofs. *Discrete Applied Mathematics*, 18(1):25–38, November 1987.

[CGS17] Kevin K. H. Cheung, Ambros M. Gleixner, and Daniel E. Steffy. Verifying integer programming results. In *Proceedings of the 19th International Conference on Integer Programming and Combinatorial Optimization (IPCO '17)*, volume 10328 of *Lecture Notes in Computer Science*, pages 148–160. Springer, June 2017.

[CHH+17] Luís Cruz-Filipe, Marijn J. H. Heule, Warren A. Hunt Jr., Matt Kaufmann, and Peter Schneider-Kamp. Efficient certified RAT verification. In *Proceedings of the 26th International Conference on Automated Deduction (CADE-26)*, volume 10395 of *Lecture Notes in Computer Science*, pages 220–236. Springer, August 2017.

[CKSW13] William Cook, Thorsten Koch, Daniel E. Steffy, and Kati Wolter. A hybrid branch-and-bound approach for exact rational mixed-integer programming. *Mathematical Programming Computation*, 5(3):305–344, September 2013.

[CMS17] Luís Cruz-Filipe, João P. Marques-Silva, and Peter Schneider-Kamp. Efficient certified resolution proof checking. In *Proceedings of the 23rd International Conference on Tools and Algorithms for the Construction*

and Analysis of Systems (TACAS '17), volume 10205 of *Lecture Notes in Computer Science*, pages 118–135. Springer, April 2017.

[CNR21] Aviad Cohen, Alexander Nadel, and Vadim Ryvchin. Local search with a sat oracle for combinatorial optimization. In Jan Friso Groote and Kim Guldstrand Larsen, editors, *Tools and Algorithms for the Construction and Analysis of Systems*, pages 87–104, Cham, 2021. Springer International Publishing.

[Coo71] Stephen A. Cook. The complexity of theorem-proving procedures. In *Proceedings of the 3rd Annual ACM Symposium on Theory of Computing (STOC '71)*, pages 151–158, May 1971.

[CR79] Stephen A. Cook and Robert A. Reckhow. The relative efficiency of propositional proof systems. *Journal of Symbolic Logic*, 44(1):36–50, March 1979. Preliminary version in *STOC '74*.

[Dar04] Adnan Darwiche. New advances in compiling CNF to decomposable negation normal form. In *Proc. of ECAI*, pages 328–332, 2004.

[DB13] Jessica Davies and Fahiem Bacchus. Exploiting the power of MIP solvers in MAXSAT. In *Proceedings of the 16th International Conference on Theory and Applications of Satisfiability Testing (SAT '13)*, volume 7962 of *Lecture Notes in Computer Science*, pages 166–181. Springer, July 2013.

[DEGH23] Jasper van Doornmalen, Leon Eifler, Ambros Gleixner, and Christopher Hojny. A proof system for certifying symmetry and optimality reasoning in integer programming. Technical Report 2311.03877, arXiv.org, November 2023.

[Dev20a] Jo Devriendt. Miplib 0-1 instances in opb format. Dataset on Zenodo, 05 2020.

[Dev20b] Jo Devriendt. Watched propagation of 0-1 integer linear constraints. In *Proceedings of the 26th International Conference on Principles and Practice of Constraint Programming (CP '20)*, volume 12333 of *Lecture Notes in Computer Science*, pages 160–176. Springer, September 2020.

[DG02] Heidi E. Dixon and Matthew L. Ginsberg. Inference methods for a pseudo-Boolean satisfiability solver. In *Proceedings of the 18th National Conference on Artificial Intelligence (AAAI '02)*, pages 635–640, July 2002.

[DGD+21] Jo Devriendt, Stephan Gocht, Emir Demirović, Jakob Nordström, and Peter Stuckey. Cutting to the core of pseudo-Boolean optimization: Combining core-guided search with cutting planes reasoning. In *Proceedings of the 35th AAAI Conference on Artificial Intelligence (AAAI '21)*, pages 3750–3758, February 2021.

[DHN+25] Simon Dold, Malte Helmert, Jakob Nordström, Gabriele Roger, and Tanja Schindler. Pseudo-Boolean proof logging for optimal classical planning. In *Proceedings of the Thirty-Fifth International Conference on Automated Planning and Scheduling*, ICAPS '25. AAAI Press, 2025.

[Die16] Reinhard Diestel. *Graph Theory*. Springer-Verlag, Heidelberg (print edition), 2016.

[DKK+26] Simon Dold, George Katsirelos, Wietze Koops, Magnus O. Myreen, Jakob Nordström, Andy Oertel, and Yong Kiam Tan. End-to-end certified graph colouring. 2026. in review.

[DLL62] Martin Davis, George Logemann, and Donald Loveland. A machine program for theorem proving. *Communications of the ACM*, 5(7):394–397, July 1962.

[DMM+24] Emir Demirović, Ciaran McCreesh, Matthew McIlree, Jakob Nordström, Andy Oertel, and Konstantin Sidorov. Pseudo-Boolean reasoning about states and transitions to certify dynamic programming and decision diagram algorithms. In *Proceedings of the 30th International Conference on Principles and Practice of Constraint Programming (CP '24)*, volume 307 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 9:1–9:21, September 2024.

[DP60] Martin Davis and Hilary Putnam. A computing procedure for quantification theory. *Journal of the ACM*, 7(3):201–215, 1960.

[EGMN20] Jan Elffers, Stephan Gocht, Ciaran McCreesh, and Jakob Nordström. Justifying all differences using pseudo-Boolean reasoning. In *Proceedings of the 34th AAAI Conference on Artificial Intelligence (AAAI '20)*, pages 1486–1494, February 2020.

[EN18] Jan Elffers and Jakob Nordström. Divide and conquer: Towards faster pseudo-Boolean solving. In *Proceedings of the 27th International Joint Conference on Artificial Intelligence (IJCAI '18)*, pages 1291–1299, July 2018.

[EPRL12] John D Eblen, Charles A Phillips, Gary L Rogers, and Michael A Langston. The maximum clique enumeration problem: algorithms, applications, and implementations. *BMC bioinformatics*, 13(Suppl 10):S5, 2012.

[ES03] Niklas Eén and Niklas Sörensson. Temporal induction by incremental SAT solving. In *Proceedings of the 1st International Workshop on Bounded Model Checking (BMC '03)*, volume 89 of *Electronic Notes in Theoretical Computer Science*, pages 543–560, July 2003.

[ES06] Niklas Eén and Niklas Sörensson. Translating pseudo-Boolean constraints into SAT. *Journal on Satisfiability, Boolean Modeling and Computation*, 2(1-4):1–26, March 2006.

[Far02] Julius Farkas. Theorie der einfachen Ungleichungen. *Journal für die Reine und Angewandte Mathematik*, 1902(124):1–27, 1902.

[FHR22] Johannes Klaus Fichte, Markus Hecher, and Valentin Roland. Proofs for propositional model counting. In *25th International Conference on Theory and Applications of Satisfiability Testing, SAT 2022, August 2-5, 2022, Haifa, Israel*, pages 30:1–30:24, 2022.

[FL23] Mathias Fleury and Peter Lammich. A more pragmatic CDCL for IsaSAT and targetting LLVM (short paper). In *Proceedings of the 29th International Conference on Automated Deduction (CADE-29)*, volume 14132 of *Lecture Notes in Computer Science*, pages 207–219. Springer, July 2023.

[FPFB24] Katalin Fazekas, Florian Pollitt, Mathias Fleury, and Armin Biere. Certifying incremental sat solving. In *LPAR 2024: Proceedings of 25th Conference on Logic for Programming, Artificial Intelligence and Reasoning, Port Louis, Mauritius*, volume 100, pages 321–340, 2024.

[FSM+24] Maarten Flippo, Konstantin Sidorov, Imko Marijnissen, Jeff Smits, and Emir Demirović. A Multi-Stage Proof Logging Framework to Certify the Correctness of CP Solvers. In Paul Shaw, editor, *30th International Conference on Principles and Practice of Constraint Programming (CP 2024)*, volume 307 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 11:1–11:20, Dagstuhl, Germany, 2024. Schloss Dagstuhl – Leibniz-Zentrum für Informatik.

[FYBH24] Nils Froleyks, Emily Yu, Armin Biere, and Keijo Heljanko. Certifying phase abstraction. In *International Joint Conference on Automated Reasoning*, pages 284–303. Springer, 2024.

[GGH22] Ambros Gleixner, Leona Gottwald, and Alexander Hoen. PaPILO: A parallel presolving library for integer and linear programming with multiprecision support. Technical Report 2206.10709, arXiv.org, June 2022.

[GGK+19] Gerald Gamrath, Ambros Gleixner, Thorsten Koch, Matthias Miltenberger, Dimitri Kniasew, Dominik Schlögel, Alexander Martin, and Dieter Weninger. Tackling industrial-scale supply chain problems by mixed-integer programming. *Journal of Computational Mathematics*, 37(6):866–888, 2019.

[GKS09] Martin Gebser, Benjamin Kaufmann, and Torsten Schaub. Solution enumeration for projected boolean search problems. In Willem-Jan van Hoeve and John N. Hooker, editors, *Integration of AI and OR Techniques in Constraint Programming for Combinatorial Optimization Problems*, pages 71–86, Berlin, Heidelberg, 2009. Springer Berlin Heidelberg.

[GMKN17] Armaël Guéneau, Magnus O. Myreen, Ramana Kumar, and Michael Norrish. Verified characteristic formulae for CakeML. In *Proceedings of the 26th European Symposium on Programming (ESOP '17)*, volume 10201 of *Lecture Notes in Computer Science*, pages 584–610. Springer, April 2017.

[GMM+20] Stephan Gocht, Ross McBride, Ciaran McCreesh, Jakob Nordström, Patrick Prosser, and James Trimble. Certifying solvers for clique and maximum common (connected) subgraph problems. In *Proceedings of the 26th International Conference on Principles and Practice of Constraint Programming (CP '20)*, volume 12333 of *Lecture Notes in Computer Science*, pages 338–357. Springer, September 2020.

[GMN20] Stephan Gocht, Ciaran McCreesh, and Jakob Nordström. Subgraph isomorphism meets cutting planes: Solving with certified solutions. In *Proceedings of the 29th International Joint Conference on Artificial Intelligence (IJCAI '20)*, pages 1134–1140, July 2020.

[GMN22] Stephan Gocht, Ciaran McCreesh, and Jakob Nordström. An auditable constraint programming solver. In *Proceedings of the 28th International Conference on Principles and Practice of Constraint Programming (CP '22)*, volume 235 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 25:1–25:18, August 2022.

[GN03] Evgueni Goldberg and Yakov Novikov. Verification of proofs of unsatisfiability for CNF formulas. In *Proceedings of the Conference on Design, Automation and Test in Europe (DATE '03)*, pages 886–891, March 2003.

[GN21] Stephan Gocht and Jakob Nordström. Certifying parity reasoning efficiently using pseudo-Boolean proofs. In *Proceedings of the 35th AAAI Conference on Artificial Intelligence (AAAI '21)*, pages 3768–3777, February 2021.

[GN22] Stephan Gocht and Jakob Nordström. Certifying parity reasoning efficiently using pseudo-Boolean proofs. Technical Report 2209.12185, arXiv.org, September 2022.

[GNY19] Stephan Gocht, Jakob Nordström, and Amir Yehudayoff. On division versus saturation in pseudo-Boolean solving. In *Proceedings of the 28th International Joint Conference on Artificial Intelligence (IJCAI '19)*, pages 1711–1718, August 2019.

[Goc22] Stephan Gocht. *Certifying Correctness for Combinatorial Algorithms by Using Pseudo-Boolean Reasoning*. PhD thesis, Lund University, June 2022. Available at `https://portal.research.lu.se/en/publications/certifying-correctness-for-combinatorial-algorithms-by-using-pseu`.

[GS19] Graeme Gange and Peter Stuckey. Certifying optimality in constraint programming. Presentation at KTH Royal Institute of Technology, February 2019.

[GSD19] Xavier Gillard, Pierre Schaus, and Yves Deville. SolverCheck: Declarative testing of constraints. In *Proceedings of the 25th International Conference on Principles and Practice of Constraint Programming (CP '19)*, volume 11802 of *Lecture Notes in Computer Science*, pages 565–582. Springer, October 2019.

[GSS21] Carla P. Gomes, Ashish Sabharwal, and Bart Selman. Model counting. In Biere et al. [BHvMW21], chapter 25, pages 993–1014.

[Hak85] Armin Haken. The intractability of resolution. *Theoretical Computer Science*, 39(2-3):297–308, August 1985.

[Heu21] Marijn JH Heule. Proofs of unsatisfiability. In Biere et al. [BHvMW21], chapter 15, pages 635–668.

[HGH23] Andrew Haberlandt, Harrison Green, and Marijn J. H. Heule. Effective Auxiliary Variables via Structured Reencoding. In Meena Mahajan and Friedrich Slivovsky, editors, *26th International Conference on Theory and Applications of Satisfiability Testing (SAT 2023)*, volume 271 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 11:1–11:19, Dagstuhl, Germany, 2023. Schloss Dagstuhl – Leibniz-Zentrum für Informatik.

[HHW14] Marijn J. H. Heule, Warren A. Hunt Jr., and Nathan Wetzler. Bridging the gap between easy generation and efficient verification of unsatisfiability proofs. *Software Testing, Verification and Reliability*, 24(8):593–607, 2014.

[HIJS24] Marijn J.H. Heule, Markus Iser, Matti Järvisalo, and Martin Suda, editors. *Proceedings of SAT Competition 2024: Solver, Benchmark and Proof Checker Descriptions*. Department of Computer Science Report Series B. Department of Computer Science, University of Helsinki, Finland, 2024.

[HKB17] Marijn J. H. Heule, Benjamin Kiesl, and Armin Biere. Short proofs without new variables. In *Proceedings of the 26th International Conference on Automated Deduction (CADE-26)*, volume 10395 of *Lecture Notes in Computer Science*, pages 130–147. Springer, August 2017.

[HKM16] Marijn J. H. Heule, Oliver Kullmann, and Victor W. Marek. Solving and verifying the boolean pythagorean triples problem via cube-and-conquer. In Nadia Creignou and Daniel Le Berre, editors, *Theory and Applications of Satisfiability Testing – SAT 2016*, pages 228–245, Cham, 2016. Springer International Publishing.

[HL06] Federico Heras and Javier Larrosa. New inference rules for efficient max-sat solving. In *AAAI*, pages 68–73, 2006.

[HPRS24] Roghayeh Hajizadeh, Tatiana Polishchuk, Elina Rönnberg, and Christiane Schmidt. A dantzig-wolfe reformulation for automated aircraft arrival scheduling in tmas. In *Proceedings of the 14th International Conference on the Practice and Theory of Automated Timetabling, PATAT 2024 :*, pages 268–271, 2024.

[HT15] Osman Hasan and Sofiene Tahar. Formal verification methods. In *Encyclopedia of Information Science and Technology, Third Edition*, pages 7162–7170. IGI Global Scientific Publishing, 2015.

[IBJ21] Hannes Ihalainen, Jeremias Berg, and Matti Järvisalo. Refined core relaxation for core-guided maxsat solving. In *27th International Conference on Principles and Practice of Constraint Programming (CP 2021)*, pages 28–1. Schloss Dagstuhl–Leibniz-Zentrum für Informatik, 2021.

[IBJ22] Hannes Ihalainen, Jeremias Berg, and Matti Järvisalo. Clause redundancy and preprocessing in maximum satisfiability. In *Proceedings of the 11th International Joint Conference on Automated Reasoning (IJCAR '22)*, volume 13385 of *Lecture Notes in Computer Science*, pages 75–94. Springer, August 2022.

[IMM19] Alexey Ignatiev, António Morgado, and João P. Marques-Silva. RC2: an efficient MaxSAT solver. *Journal on Satisfiability, Boolean Modeling and Computation*, 11(1):53–64, September 2019.

[JBBJ25] Christoph Jabs, Jeremias Berg, Bart Bogaerts, and Matti Järvisalo. Certifying pareto-optimality in multi objective maximum satisfiability. In Arie Gurfinkel and Marijn Heule, editors, *Tools and Algorithms for the Construction and Analysis of Systems*, pages 108–129, Cham, 2025. Springer Nature Switzerland.

[JHB12] Matti Järvisalo, Marijn J. H. Heule, and Armin Biere. Inprocessing rules. In *Proceedings of the 6th International Joint Conference on Automated Reasoning (IJCAR '12)*, volume 7364 of *Lecture Notes in Computer Science*, pages 355–370. Springer, June 2012.

[JMM15] Saurabh Joshi, Ruben Martins, and Vasco M. Manquinho. Generalized totalizer encoding for pseudo-Boolean constraints. In *Proceedings of the 21st International Conference on Principles and Practice of Constraint Programming (CP '15)*, volume 9255 of *Lecture Notes in Computer Science*, pages 200–209. Springer, August-September 2015.

[KBSJ17] Tuukka Korhonen, Jeremias Berg, Paul Saikko, and Matti Järvisalo. Maxpre: an extended maxsat preprocessor. In Serge Gaspers and Toby Walsh, editors, *Proceedings of the 20th International Conference on Theory and Applications of Satisfiability Testing, (SAT '17)*, volume 10491 of *Lecture Notes in Computer Science*, pages 449–456. Springer, 2017.

[KLBM+25] Wietze Koops, Daniel Le Berre, Magnus O. Myreen, Jakob Nordström, Andy Oertel, Yong Kiam Tan, and Marc Vinyals. Practically Feasible Proof Logging for Pseudo-Boolean Optimization. In Maria Garcia de la Banda, editor, *31st International Conference on Principles and Practice of Constraint Programming (CP 2025)*, volume 340 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 21:1–21:27, Dagstuhl, Germany, 2025. Schloss Dagstuhl – Leibniz-Zentrum für Informatik.

[KMMS06] Dieter Kratsch, Ross M. McConnell, Kurt Mehlhorn, and Jeremy P. Spinrad. Certifying algorithms for recognizing interval graphs and permutation graphs. *SIAM Journal on Computing*, 36(2):326–353, 2006.

[KMTM18] Ramana Kumar, Eric Mullen, Zachary Tatlock, and Magnus O. Myreen. Software verification with ITPs should use binary code extraction to reduce the TCB. In *Proceedings of the 9th International Conference on Interactive Theorem Proving (ITP '18)*, volume 10895 of *Lecture Notes in Computer Science*, pages 362–369. Springer, July 2018.

[Kra19] Jan Krajíček. *Proof Complexity*, volume 170 of *Encyclopedia of Mathematics and Its Applications*. Cambridge University Press, March 2019.

[KRH18] Benjamin Kiesl, Adrián Rebola-Pardo, and Marijn J. H. Heule. Extended resolution simulates DRAT. In *Proceedings of the 9th International Joint Conference on Automated Reasoning (IJCAR '18)*, volume 10900 of *Lecture Notes in Computer Science*, pages 516–531. Springer, July 2018.

[KT24] Leszek Kołodziejczyk and Neil Thapen. The strength of the dominance rule. In *Proceedings of the 27th International Conference on Theory and Applications of Satisfiability Testing (SAT '24)*, volume 305 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 20:1–20:22, August 2024.

[KZFH12] Miyuki Koshimura, Tong Zhang, Hiroshi Fujita, and Ryuzo Hasegawa. Qmaxsat: A partial max-sat solver: System description. *Journal on Satisfiability, Boolean Modelling and Computation*, 8(1-2):95–100, 2012.

[Lam20] Peter Lammich. Efficient verified (UN)SAT certificate checking. *Journal of Automated Reasoning*, 64(3):513–532, March 2020. Extended version of paper in *CADE* 2017.

[Lam24] Peter Lammich. Fast and verified unsat certificate checking. In Christoph Benzmüller, Marijn J.H. Heule, and Renate A. Schmidt, editors, *Automated Reasoning*, pages 439–457, Cham, 2024. Springer Nature Switzerland.

[Lev73] Leonid A. Levin. Universal sequential search problems. *Problemy peredachi informatsii*, 9(3):115–116, 1973. In Russian. Available at `http://mi.mathnet.ru/ppi914`.

[LP10] Daniel Le Berre and Anne Parrain. The Sat4j library, release 2.2. *Journal on Satisfiability, Boolean Modeling and Computation*, 7:59–64, July 2010.

[LXC+21] Chu-Min Li, Zhenxing Xu, Jordi Coll, Felip Manyà, Djamal Habet, and Kun He. Combining clause learning and branch and bound for MaxSAT. In *Proceedings of the 27th International Conference on Principles and Practice of Constraint Programming (CP '21)*, volume 210 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 38:1–38:18, October 2021.

[MAGL11] João Marques-Silva, Josep Argelich, Ana Graça, and Inês Lynce. Boolean lexicographic optimization: algorithms & applications. *Annals of Mathematics and Artificial Intelligence*, 62(3-4):317–343, 2011.

[Mar10] João P. Marques-Silva. Minimal unsatisfiability: Models, algorithms and applications (Invited paper). In *Proceedings of the 40th IEEE International Symposium on Multiple-Valued Logic*, pages 9–14, May 2010.

[Max22] MaxSAT evaluation 2022. `https://maxsat-evaluations.github.io/2022`, August 2022.

[Max23] MaxSAT evaluation 2023. `https://maxsat-evaluations.github.io/2023`, July 2023.

[MDM14] António Morgado, Carmine Dodaro, and João P. Marques-Silva. Core-guided MaxSAT with soft cardinality constraints. In *Proceedings of the 20th International Conference on Principles and Practice of Constraint Programming (CP '14)*, volume 8656 of *Lecture Notes in Computer Science*, pages 564–573. Springer, September 2014.

[MJML14] Ruben Martins, Saurabh Joshi, Vasco M. Manquinho, and Inês Lynce. Incremental cardinality constraints for MaxSAT. In *Proceedings of the 20th International Conference on Principles and Practice of Constraint Programming (CP '14)*, volume 8656 of *Lecture Notes in Computer Science*, pages 531–548. Springer, September 2014.

[MKL+95] Barton P Miller, David Koski, Cjin Pheow Lee, Vivekandanda Maganty, Ravi Murthy, Ajitkumar Natarajan, and Jeff Steidl. Fuzz revisited: A re-examination of the reliability of unix utilities and services. Technical report, University of Wisconsin-Madison Department of Computer Sciences, 1995.

[MM23] Matthew McIlree and Ciaran McCreesh. Proof logging for smart extensional constraints. In *Proceedings of the 29th International Conference on Principles and Practice of Constraint Programming (CP '23)*,

volume 280 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 26:1–26:17, August 2023.

[MM25] Matthew McIlree and Ciaran McCreesh. Certifying bounds propagation for integer multiplication constraints. In *Proceedings of the 39th AAAI Conference on Artificial Intelligence (AAAI '25)*, pages 11309–11317, February-March 2025.

[MML14] Ruben Martins, Vasco M. Manquinho, and Inês Lynce. Open-WBO: A modular MaxSAT solver. In *Proceedings of the 17th International Conference on Theory and Applications of Satisfiability Testing (SAT '14)*, volume 8561 of *Lecture Notes in Computer Science*, pages 438–445. Springer, July 2014.

[MMN24] Matthew McIlree, Ciaran McCreesh, and Jakob Nordström. Proof logging for the circuit constraint. In *Proceedings of the 21st International Conference on the Integration of Constraint Programming, Artificial Intelligence, and Operations Research (CPAIOR '24)*, volume 14743 of *Lecture Notes in Computer Science*, pages 38–55. Springer, May 2024.

[MMNS11] Ross M. McConnell, Kurt Mehlhorn, Stefan Näher, and Pascal Schweitzer. Certifying algorithms. *Computer Science Review*, 5(2):119–161, May 2011.

[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 Design Automation Conference (DAC '01)*, pages 530–535, June 2001.

[MN89] Kurt Mehlhorn and Stefan Näher. Leda a library of efficient data types and algorithms. In Antoni Kreczmar and Grazyna Mirkowska, editors, *Mathematical Foundations of Computer Science 1989*, pages 88–106, Berlin, Heidelberg, 1989. Springer Berlin Heidelberg.

[MN95] Kurt Mehlhorn and Stefan Näher. Leda: a platform for combinatorial and geometric computing. *Commun. ACM*, 38(1):96–102, January 1995.

[MO12] David F. Manlove and Gregg O'Malley. Paired and altruistic kidney donation in the UK: Algorithms and experimentation. In *Proceedings of the 11th International Symposium on Experimental Algorithms (SEA '12)*, volume 7276 of *Lecture Notes in Computer Science*, pages 271–282. Springer, June 2012.

[MO14] Magnus O. Myreen and Scott Owens. Proof-producing translation of higher-order logic into pure and stateful ML. *Journal of Functional Programming*, 24(2–3):284–315, January 2014.

[MOS15] Renata Mansini, Włodzimierz Ogryczak, and M. Grazia Speranza. *Linear and mixed integer programming for portfolio optimization*, volume 21. Springer, 2015.

[MS99] João P. Marques-Silva and Karem A. Sakallah. GRASP: A search algorithm for propositional satisfiability. *IEEE Transactions on Computers*, 48(5):506–521, May 1999. Preliminary version in *ICCAD '96*.

[MSB11] Glenford J Myers, Corey Sandler, and Tom Badgett. *The art of software testing*. John Wiley & Sons, 2011.

[MSLM21] Joao Marques-Silva, Ines Lynce, and Sharad Malik. Conflict-driven clause learning sat solvers. In Biere et al. [BHvMW21], chapter 4, pages 133–182.

[Mur68] Katta G. Murty. An algorithm for ranking all the assignments in order of increasing cost. *Operations Research*, 16(3):682–687, 1968.

[NB14] Nina Narodytska and Fahiem Bacchus. Maximum satisfiability using core-guided maxsat resolution. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 28, 2014.

[Nie00] Jürg Nievergelt. Exhaustive search, combinatorial optimization and enumeration: Exploring the potential of raw computing power. In Václav Hlaváč, Keith G. Jeffery, and Jiří Wiedermann, editors, *SOFSEM 2000: Theory and Practice of Informatics*, pages 18–35, Berlin, Heidelberg, 2000. Springer Berlin Heidelberg.

[NORZ24] Robert Nieuwenhuis, Albert Oliveras, Enric Rodríguez-Carbonell, and Rui Zhao. Speeding up pseudo-Boolean propagation. In *Proceedings of the 27th International Conference on Theory and Applications of Satisfiability Testing (SAT '24)*, volume 305 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 22:1–22:18, August 2024.

[NWP02] Tobias Nipkow, Markus Wenzel, and Lawrence C Paulson. *Isabelle/HOL: a proof assistant for higher-order logic*. Springer, 2002.

[PB23] Tobias Paxian and Armin Biere. Uncovering and classifying bugs in MaxSAT solvers through fuzzing and delta debugging. In *Proceedings of the 14th International Workshop on Pragmatics of SAT*, volume 3545 of *CEUR Workshop Proceedings*, pages 59–71. CEUR-WS.org, July 2023.

[PB26] Tobias Paxian and Armin Biere. Maxsat fuzzing and delta debugging. *Journal of Artificial Intelligence Research*, 85, 2026.

[PFB23] Florian Pollitt, Mathias Fleury, and Armin Biere. Faster LRAT checking than solving with CaDiCaL. In *Proceedings of the 26th International Conference on Theory and Applications of Satisfiability Testing (SAT '23)*, volume 271 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 21:1–21:12, July 2023.

[PR16] Tobias Philipp and Adrián Rebola-Pardo. DRAT proofs for XOR reasoning. In *Proceedings of the 15th European Conference on Logics in Artificial Intelligence (JELIA '16)*, volume 10021 of *Lecture Notes in Computer Science*, pages 415–429. Springer, November 2016.

[PRB18] Tobias Paxian, Sven Reimer, and Bernd Becker. Dynamic polynomial watchdog encoding for solving weighted MaxSAT. In *Proceedings of the 21st International Conference on Theory and Applications of Satisfiability Testing (SAT '18)*, volume 10929 of *Lecture Notes in Computer Science*, pages 37–53. Springer, July 2018.

[PRB21] Tobias Paxian, Pascal Raiola, and Bernd Becker. On preprocessing for weighted MaxSAT. In *Proceedings of the 22nd International Conference on Verification, Model Checking, and Abstract Interpretation (VMCAI '21)*, volume 12597 of *Lecture Notes in Computer Science*, pages 556–577. Springer, January 2021.

[PS15] Tobias Philipp and Peter Steinke. PBLib – a library for encoding pseudo-Boolean constraints into CNF. In *Proceedings of the 18th International Conference on Theory and Applications of Satisfiability Testing (SAT '15)*, volume 9340 of *Lecture Notes in Computer Science*, pages 9–16. Springer, September 2015.

[Pse16] Pseudo-Boolean competition 2016. `https://www.cril.univ-artois.fr/PB16/`, July 2016.

[Pse24] Pseudo-Boolean competition 2024. `https://www.cril.univ-artois.fr/PB24/`, August 2024.

[RM21] Olivier Roussel and Vasco Manquinho. Pseudo-boolean and cardinality constraints. In Biere et al. [BHvMW21], chapter 28, pages 1087–1129.

[Rob65] John Alan Robinson. A machine-oriented logic based on the resolution principle. *Journal of the ACM*, 12(1):23–41, January 1965.

[RP23] Adrián Rebola-Pardo. Even Shorter Proofs Without New Variables. In Meena Mahajan and Friedrich Slivovsky, editors, *26th International Conference on Theory and Applications of Satisfiability Testing (SAT 2023)*, volume 271 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 22:1–22:20, Dagstuhl, Germany, 2023. Schloss Dagstuhl – Leibniz-Zentrum für Informatik.

[RS18] Adrián Rebola-Pardo and Martin Suda. A theory of satisfiability-preserving proofs in sat solving. In *LPAR*, pages 583–603, 2018.

[RvBW06] Francesca Rossi, Peter van Beek, and Toby Walsh, editors. *Handbook of Constraint Programming*, volume 2 of *Foundations of Artificial Intelligence*. Elsevier, 2006.

[Rys63] Herbert John Ryser. *Combinatorial mathematics*, volume 14. American Mathematical Soc., 1963.

[Sak21] Karem A. Sakallah. Symmetry and satisfiability. In Biere et al. [BHvMW21], chapter 13, pages 509–570.

[SAT13] SAT competition 2013. `https://satisfiability.org/competition/2013/`, July 2013.

[Sau24] J. Sauppe, editor. *Mathematical Programming Glossary*. INFORMS Computing Society, `http://glossary.informs.org`, 2006–24. Originally authored by Harvey J. Greenberg, 1999-2006.

[Sch05] Alexander Schrijver. On the history of combinatorial optimization (till 1960). In K. Aardal, G.L. Nemhauser, and R. Weismantel, editors, *Discrete Optimization*, volume 12 of *Handbooks in Operations Research and Management Science*, pages 1–68. Elsevier, 2005.

[Sch20] Philine Schiewe. *Integrated optimization in public transport planning*, volume 160. Springer, 2020.

[SH23] Bernardo Subercaseaux and Marijn J. H. Heule. The packing chromatic number of the infinite square grid is 15. In Sriram Sankaranarayanan and Natasha Sharygina, editors, *Tools and Algorithms for the Construction and Analysis of Systems*, pages 389–406, Cham, 2023. Springer Nature Switzerland.

[Sin05] Carsten Sinz. Towards an optimal CNF encoding of Boolean cardinality constraints. In *Proceedings of the 11th International Conference on Principles and Practice of Constraint Programming (CP '05)*, volume 3709 of *Lecture Notes in Computer Science*, pages 827–831. Springer, October 2005.

[SIY+20] Ryosuke Shibukawa, Shoichi Ishida, Kazuki Yoshizoe, Kunihiro Wasa, Kiyosei Takasu, Yasushi Okuno, Kei Terayama, and Koji Tsuda. Compret: a comprehensive recommendation framework for chemical synthesis planning with algorithmic enumeration. *Journal of cheminformatics*, 12(1):52, 2020.

[SN15] Masahiko Sakai and Hidetomo Nabeshima. Construction of an ROBDD for a PB-constraint in band form and related techniques for PB-solvers. *IEICE Transactions on Information and Systems*, 98-D(6):1121–1127, June 2015.

[SRB25] Dominik Schreiber, Niccolò Rigi-Luperti, and Armin Biere. Streamlining Distributed SAT Solver Design. In Jeremias Berg and Jakob Nordström, editors, *28th International Conference on Theory and Applications of Satisfiability Testing (SAT 2025)*, volume 341 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 27:1–27:23, Dagstuhl, Germany, 2025. Schloss Dagstuhl – Leibniz-Zentrum für Informatik.

[TD20] Rodrigue Konan Tchinda and Clémentin Tayou Djamégni. On certifying the UNSAT result of dynamic symmetry-handling-based SAT solvers. *Constraints*, 25(3–4):251–279, December 2020.

[THM23] Yong Kiam Tan, Marijn J. H. Heule, and Magnus O. Myreen. Verified propagation redundancy and compositional UNSAT checking in CakeML. *International Journal on Software Tools for Technology Transfer*, 25:167–184, February 2023. Preliminary version in *TACAS '21*.

[TMK+19] Yong Kiam Tan, Magnus O. Myreen, Ramana Kumar, Anthony C. J. Fox, Scott Owens, and Michael Norrish. The verified CakeML compiler backend. *Journal of Functional Programming*, 29:e2:1–e2:57, February 2019.

[Tse68] Grigori Tseitin. On the complexity of derivation in propositional calculus. In A. O. Silenko, editor, *Structures in Constructive Mathematics and Mathematical Logic, Part II*, pages 115–125. Consultants Bureau, New York-London, 1968.

[TYS+24] Yong Kiam Tan, Jiong Yang, Mate Soos, Magnus O. Myreen, and Kuldeep S. Meel. Formally certified approximate model counting. In Arie Gurfinkel and Vijay Ganesh, editors, *Computer Aided Verification - 36th International Conference, CAV 2024, Montreal, QC, Canada, July 24-27, 2024, Proceedings, Part I*, volume 14681 of *Lecture Notes in Computer Science*, pages 153–177. Springer, 2024.

[Van08] Allen Van Gelder. Verifying RUP proofs of propositional unsatisfiability. In *10th International Symposium on Artificial Intelligence and Mathematics (ISAIM '08)*, 2008. Available at `http://isaim2008.unl.edu/index.php?page=proceedings`.

[VDB22] Dieter Vandesande, Wolf De Wulf, and Bart Bogaerts. QMaxSATpb: A certified MaxSAT solver. In *Proceedings of the 16th International Conference on Logic Programming and Non-monotonic Reasoning (LPNMR '22)*, volume 13416 of *Lecture Notes in Computer Science*, pages 429–442. Springer, September 2022.

[VG02] Allen Van Gelder. Extracting (easily) checkable proofs from a satisfiability solver that employs both preorder and postorder resolution. In *7th International Symposium on AI and Mathematics*, 2002.

[VS10] Michael Veksler and Ofer Strichman. A proof-producing CSP solver. In *Proceedings of the 24th AAAI Conference on Artificial Intelligence (AAAI '10)*, pages 204–209, July 2010.

[Wal96] Mark Wallace. Practical applications of constraint programming. *Constraints*, 1(1):139–168, 1996.

[War98] Joost P. Warners. A linear-time transformation of linear inequalities into conjunctive normal form. *Information Processing Letters*, 68(2):63–69, October 1998.

[Wei25] Nils Weidmann. Multi-League Sports Scheduling with Team Interdependencies: An Optimization Model. In Maria Garcia de la Banda, editor, *31st International Conference on Principles and Practice*

of Constraint Programming (CP 2025), volume 340 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 37:1–37:19, Dagstuhl, Germany, 2025. Schloss Dagstuhl – Leibniz-Zentrum für Informatik.

[WHH14] Nathan Wetzler, Marijn J. H. Heule, and Warren A. Hunt Jr. DRAT-trim: Efficient checking and trimming using expressive clausal proofs. In *Proceedings of the 17th International Conference on Theory and Applications of Satisfiability Testing (SAT '14)*, volume 8561 of *Lecture Notes in Computer Science*, pages 422–429. Springer, July 2014.

[WS24] Dominik Winterer and Zhendong Su. Validating smt solvers for correctness and performance via grammar-based enumeration. *Proc. ACM Program. Lang.*, 8(OOPSLA2), October 2024.

[YBH21] Emily Yu, Armin Biere, and Keijo Heljanko. Progress in certifying hardware model checking results. In *International Conference on Computer Aided Verification*, pages 363–386. Springer, 2021.

[ZWCX22] Xiaogang Zhu, Sheng Wen, Seyit Camtepe, and Yang Xiang. Fuzzing: A survey for roadmap. *ACM Comput. Surv.*, 54(11s), September 2022.

Included Papers

Certified CNF Translations for Pseudo-Boolean Solving

Abstract

The dramatic improvements in Boolean satisfiability (SAT) solving since the turn of the millennium have made it possible to leverage state-of-the-art conflict-driven clause learning (CDCL) solvers for many combinatorial problems in academia and industry, and the use of proof logging has played a crucial role in increasing the confidence that the results these solvers produce are correct. However, the fact that SAT proof logging is performed in conjunctive normal form (CNF) clausal format means that it has not been possible to extend guarantees of correctness to the use of SAT solvers for more expressive combinatorial paradigms, where the first step is an unverified translation of the input to CNF.

In this work, we show how cutting-planes-based reasoning can provide proof logging for solvers that translate pseudo-Boolean (a.k.a. 0-1 integer linear) decision problems to CNF and then run CDCL. To support a wide range of encodings, we provide a uniform and easily extensible framework for proof logging of CNF translations. We are hopeful that this is just a first step towards providing a unified proof logging approach that will also extend to maximum satisfiability (MaxSAT) solving and pseudo-Boolean optimization in general.

1 Introduction

Boolean satisfiability (SAT) solving has witnessed striking improvements over the last couple of decades, starting with the introduction of *conflict-driven clause learning (CDCL)* SAT solvers [MS99, MMZ⁺01], and this has led to a wide range of applications including large-scale problems in both academia and industry [BHvMW21]. The conflict-driven paradigm has also been successfully exported to other areas such as *maximum satisfiability (MaxSAT)*, *pseudo-Boolean (PB) solving*, *constraint programming (CP)*, and *mixed integer linear programming (MIP)*. As modern combinatorial

Stephan Gocht, Jakob Nordström, Ruben Martins, and Andy Oertel. “Certified CNF Translations for Pseudo-Boolean Solving”. Accepted for publication in *Journal of Artificial Intelligence Research*. Preliminary version in *Proceedings of the 25th International Conference on Theory and Applications of Satisfiability Testing (SAT ’22)*, volume 236 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 16:1–16:25, August 2022.

solvers are used to attack ever more challenging problems, and employ ever more sophisticated heuristics and optimizations to do so, the question arises whether we can trust the results they produce. Sadly, it is well documented that state-of-the-art CP and MIP solvers can return incorrect solutions [AGJ+18, CKSW13, GSD19]. For SAT solvers, however, analogous problems [BLB10] have been successfully addressed by the introduction of *proof logging*, requiring that solvers should be *certifying* [MMNS11] in the sense that they output machine-verifiable proofs of their claims that can be verified by a stand-alone *proof checker*.

A number of different proof logging formats have been developed for SAT solving, including *RUP* [GN03, Van08], *TraceCheck* [Bie06], *DRAT* [HHW13a, HHW13b, WHH14], *GRIT* [CFMSSK17], and *LRAT* [CFHH+17]. Since 2013, the SAT competitions [BBHJ13] require solvers to be certifying, with *DRAT* established as the standard format. It would be highly desirable to have such proof logging also for stronger combinatorial solving paradigms, but while methods such as *DRAT* are extremely powerful in theory, the limitation to a clausal format makes it hard to capture more advanced forms of reasoning in a succinct way. A more fundamental concern is that it is not clear how these proof logging methods should deal with input that is not presented in conjunctive normal form (CNF). One way to address this problem could be to allow extensions to the *DRAT* format [BCH21]. However, we focus on another approach pursued in recent years to develop stronger proof logging methods based on more expressive formalisms such as binary decision diagrams [BB21], algebraic reasoning [KBBN22, KB21, KFB20, RBK+18], pseudo-Boolean reasoning [EGMN20, GMM+20, GMN20, GN21, BGMN22, GMN22], and integer linear programming [CGS17, EG21].

Our Contribution In this work, we consider the use of CDCL for pseudo-Boolean solving, where the pseudo-Boolean input (i.e., a 0-1 integer linear program) is translated to CNF and passed to a SAT solver, as pioneered in MiniSat+ [ES06]. The two solvers NaPS [SN15] and Open-WBO [MML14] using this approach were among the top performers in the latest pseudo-Boolean evaluation in 2016. While *DRAT* proof logging can certify unsatisfiability of the translated formula, it cannot prove correctness of the translation, not only since there is no known method of carrying out PB reasoning efficiently in *DRAT* (except for constraints with small coefficients [BBH22]), but also, and more fundamentally, because the input is not in CNF.

We demonstrate how to instead use the *cutting planes* proof method [CCT87], enhanced with a rule for introducing extension variables [GN21], to show that the CNF formula resulting from the translation can be derived from the original pseudo-Boolean constraints. Since this method is a strict extension of *DRAT*, we can combine the proof for the translation with the SAT solver *DRAT* proof log (with appropriate syntactic modifications). In this way we achieve end-to-end verification of the pseudo-Boolean solving process using the proof checker VeriPB [GN21, BGMN22] as illustrated in Figure 1. We note that verifying the correctness of the pseudo-Boolean encoding for the problem is beyond the scope of this paper.

One challenge when certifying PB-to-CNF translations is that there are many different ways of encoding pseudo-Boolean constraints into CNF (as catalogued

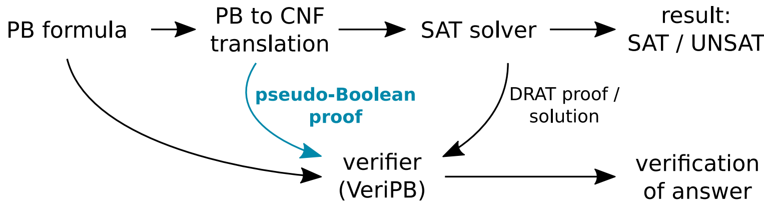


Figure 1: *Proof logging workflow for pseudo-Boolean solving with our contribution highlighted in blue boldface.*

in, e.g., [PS15]), and it is time-consuming (and error-prone) to code up proof logging for every single encoding. However, many of the encodings can be understood as first designing a circuit to evaluate whether the PB constraint is satisfied, and then writing down a CNF formula enforcing the computation of this circuit. An important part of our contribution is that we develop a general proof logging method for a wide class of such circuits. The pseudo-Boolean format used for proof logging makes it easy to derive 0-1 linear inequalities describing the circuit computations, and once this has been done the desired clauses in the CNF translation can simply be obtained by so-called *reverse unit propagation (RUP)* [GN03, Van08], obviating the need for complicated syntactic proofs. We apply this method to the *sequential counter* [Sin05], *totalizer* [BB03], *generalized totalizer* [JMM15] and *binary adder network* [ES06, War98] encodings, and report results from an empirical evaluation of the efficiency of proof generation and verification. As an additional application, we show how our certified PB-to-CNF translations can be combined with SAT proof logging to certify, for the first time, the correctness of claimed optimal values for instances in the MaxSAT Evaluation 2022.

We note that a stronger result than certifying that the CNF translation can be derived from the pseudo-Boolean input would be to certify *equivalence* of the original pseudo-Boolean formula F and the translated CNF formula F', in the sense that (a) any satisfying assignment α to F could be extended to an assignment α' also to the new variables introduced during translation that would satisfy F', and that (b) any satisfying assignment α' to F' also has to satisfiy F. The tools we develop can reach this more ambitious goal in principle, but since some additional technical problems arise along the way we have to leave this as future work.

Outline of This Paper After discussing preliminaries in Section 2, we illustrate our method for the sequential counter encoding in Section 3. Section 4 presents the general framework, and we discuss how to apply it to adder networks in Section 5 and (generalized) totalizer encoding in Section 6. We report data from our experimental evaluation in Section 7 and conclude with a discussion of some directions for future research in Section 8.

2 Preliminaries

Let us start with a review of some standard material that can also be found in, e.g., [BN21, GN21]. A *literal* ℓ over a Boolean variable x is x itself or its negation $\overline{x}$, where variables can be assigned values 0 (false) or 1 (true), so that $\overline{x} = 1 - x$. For notational convenience, we define $\overline{\overline{x}} \doteq x$ (where we use $\doteq$ to denote syntactic equality). We write $[n] = \{1, 2, \ldots, n\}$ to denote the n first positive integers, and sometimes write $\vec{x} = \{x_i \mid i \in [n]\}$ to denote a set of variables, where the size n of the set is understood from context (or is not important). A *pseudo-Boolean (PB) constraint* is a 0-1 linear inequality

$$C \;\doteq\; \textstyle\sum_i a_i \ell_i \geq A\,, \tag{1}$$

which without loss of generality we always assume to be in *normalized form* [Bar95]; i.e., all literals ℓ_i are over distinct variables and the coefficients a_i and the *degree (of falsity)* A are non-negative integers. The normalized form of the *negation* of C in (1) is the constraint

$$\neg C \;\doteq\; \textstyle\sum_i a_i \overline{\ell}_i \geq \sum_i a_i - A + 1 \tag{2}$$

(encoding that the sum of the coefficients of falsified literals in C is so large that coefficients of satisfied literals can contribute at most $A - 1$). We use equality constraints

$$C \doteq \textstyle\sum_i a_i \ell_i = A \tag{3a}$$

as syntactic sugar for the pair of inequalities

$$C^{\Rightarrow} \doteq \textstyle\sum_i a_i \ell_i \geq A \tag{3b}$$

and

$$C^{\Leftarrow} \doteq \textstyle\sum_i -a_i \ell_i \geq -A \tag{3c}$$

(with the latter converted to normalized form). We write $\sum_i a_i \ell_i \bowtie A$ for $\bowtie \in \{\geq, \leq, =\}$ for constraints that are either inequalities or equalities. A *pseudo-Boolean formula* is a conjunction $F = \bigwedge_j C_j$ of PB constraints. A *cardinality constraint* is a PB constraint with all coefficients equal to 1. If the degree is also 1, then the constraint

$$\ell_1 + \cdots + \ell_k \geq 1 \tag{4a}$$

is equivalent to the *(disjunctive) clause*

$$\ell_1 \lor \cdots \lor \ell_k\,, \tag{4b}$$

and so CNF formulas are just special cases of pseudo-Boolean formulas.

A *(partial) assignment* ρ is a (partial) function from variables to $\{0, 1\}$, which we extend to literals by respecting the meaning of negation. Applying ρ to a constraint C as in (1) yields the constraint $C\!\restriction_\rho$ obtained by substituting values for all assigned variables, shifting constants to the right-hand side, and adjusting the degree appropriately, and for a formula F we define $F\!\restriction_\rho = \bigwedge_j C_j\!\restriction_\rho$. The

constraint C is *satisfied* by ρ if $\sum_{\rho(\ell_i)=1} a_i \geq A$ (or, equivalently, if the restricted constraint $C\restriction_\rho$ has a non-positive degree and is thus trivial). An assignment ρ satisfies $F \doteq \bigwedge_j C_j$ if it satisfies all C_j, in which case F is *satisfiable*. A formula without satisfying assignments is *unsatisfiable*. Two formulas are *equisatisfiable* if they are both satisfiable or both unsatisfiable.

Cutting planes as defined in [CCT87] is a method for iteratively deriving new constraints C implied by a PB formula F. If C and D are previously derived constraints, or are *axiom constraints* in F, then any positive integer *linear combination* of these constraints can be derived. (By a linear combination of two equality constraints C and D, we mean the identical linear combinations of $C^{\Rightarrow}$ and $D^{\Rightarrow}$ and of $C^{\Leftarrow}$ and $D^{\Leftarrow}$, respectively.) We can also add *literal axioms* $\ell_i \geq 0$ to a previously derived constraint. For a constraint $\sum_i a_i \cdot \ell_i \geq A$ in normalized form, we can use *division* by a positive integer d to derive $\sum_i \lceil a_i/d \rceil \ell_i \geq \lceil A/d \rceil$, dividing and rounding up the degree and coefficients, and it is sometimes convenient to also include a *saturation* rule deriving $\sum_i \min\{a_i, A\} \cdot \ell_i \geq A$ from $\sum_i a_i \cdot \ell_i \geq A$. We remark that the soundness of the division and saturation rules as stated depends on the constraints being presented in normalized form.

For PB formulas F, F' and constraints C, C', we say that F *implies* or *models* C, denoted $F \models C$, if any assignment satisfying F must also satisfy C, and we write $F \models F'$ if $F \models C'$ for all $C' \in F'$. It is clear that any collection of constraints F' derived (iteratively) from F by cutting planes are implied in this sense, and cutting planes is an *implicationally complete* method in the sense that any implied constraint can also be derived syntactically.

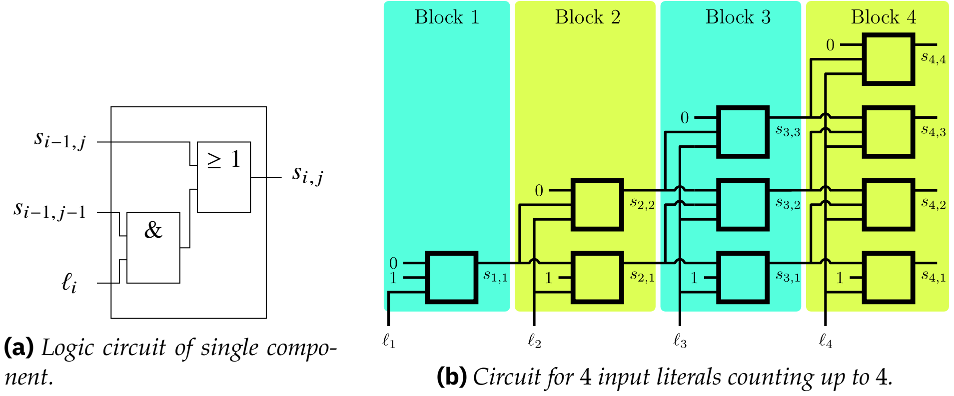
A constraint C is said to *unit propagate* the literal ℓ under ρ if $C\restriction_\rho$ cannot be satisfied unless ℓ is set to true. During *unit propagation* on F under ρ, we extend ρ iteratively by assignments to any propagated literals until an assignment ρ' is reached under which no constraint $C \in F$ is propagating, or under which some constraint C propagates a literal that has already been assigned to the opposite value. The latter scenario is called a *conflict*, since ρ' *violates* the constraint C in this case. We say that F implies C by *reverse unit propagation (RUP)*, and that C is a *RUP constraint* with respect to F, if $F \wedge \neg C$ unit propagates to conflict under the empty assignment. It is not hard to see that $F \models C$ holds if C is a RUP constraint, but the opposite direction is not necessarily true.

For introducing new variables, we will use the *reification* rule saying that we can introduce the *reified constraints*

$$z \Rightarrow \textstyle\sum_i a_i \ell_i \geq A \quad \doteq \quad A\overline{z} + \sum_i a_i \ell_i \geq A \tag{5a}$$

$$z \Leftarrow \textstyle\sum_i a_i \ell_i \geq A \quad \doteq \quad \left(\sum_i a_i - A + 1\right) \cdot z + \sum_i a_i \overline{\ell}_i \geq \sum_i a_i - A + 1 \tag{5b}$$

provided that z is a *fresh variable* that is not in the formula and has not appeared previously in the derivation. A moment of thought reveals that the constraint (5a) says that if z is true, then $\sum_i a_i \ell_i \geq A$ has to hold, and this explains the notation $z \Rightarrow \sum_i a_i \ell_i \geq A$ introduced for this constraint. In an analogous fashion, the constraint (5b) says that if $\sum_i a_i \ell_i \geq A$ holds, then z has to be true. We will write $z \Leftrightarrow \sum_i a_i \ell_i \geq A$ for the conjunction of the constraints (5a) and (5b). Adding such reification constraints preserves equisatisfiability, since any satisfying assignment to F can be extended by setting the fresh variable z as required to satisfy the



(a) *Logic circuit of single component.*

(b) *Circuit for 4 input literals counting up to 4.*

Figure 2: *Circuit representation of the sequential counter encoding.*

implications. The reification rule is a special case of the redundance rule in [GN21], where we can add any *redundant* constraint C with the property that F and $F \wedge D$ are equisatisfiable.

3 Certified CNF Translation Using the Sequential Counter Encoding

To give a concrete illustration of our approach for proving the correctness of translations of pseudo-Boolean constraints, in this section we consider how to convert cardinality constraints $\sum_{i=1}^{n} \ell_i \bowtie k$ to CNF using the *sequential counter* encoding [Sin05]. This encoding is based on a circuit summing up the input bits one by one, with intermediate variables $s_{i,j}$ for $i \in [n]$ and $j \in [i]$ evaluating to true if and only if $\sum_{t=1}^{i} \ell_t \geq j$ holds. The variables $s_{i,j}$ can be computed inductively as in Figure 2a by the formula

$$s_{i,j} \leftrightarrow \big((\ell_i \wedge s_{i-1,j-1}) \vee s_{i-1,j}\big) \tag{6}$$

saying that $s_{i,j}$ is true either if the first $i-1$ literals add up to $j-1$ and the ith literal is true, or if already the first $i-1$ literals add up to j. The circuit constructed in this way, shown in Figure 2b, can be partitioned into n blocks, where the ith block computes the variables $s_{i,j}$ for $j \in [i]$ from the ith input bit ℓ_i and the variables $s_{i-1,j}$ in the previous block. Identifying such blocks in the circuit is a key component in our method for proving that the CNF translation is correct.

For the sequential counter circuit, we obtain the CNF encoding of the constraint $\sum_{i=1}^{n} \ell_i \bowtie k$ by translating each component in Figure 2a (as described by Equation (6)) to the clausal constraints

$$\overline{\ell}_i + \overline{s}_{i-1,j-1} + s_{i,j} \geq 1 \tag{7a}$$

$$\overline{s}_{i-1,j} + s_{i,j} \geq 1 \tag{7b}$$

$$\ell_i + s_{i-1,j} + \overline{s}_{i,j} \geq 1 \tag{7c}$$

$$s_{i-1,j-1} + \overline{s}_{i,j} \geq 1 \tag{7d}$$

for $i \in [n]$ and $j \in [i]$. For all i we set $s_{i,0} = 1$ and simplify, so that constraint (7a) turns into $\overline{\ell}_i + s_{i,1} \geq 1$ and constraint (7d) is satisfied and disappears. We also set $s_{i-1,i} = 0$, so that (7c) becomes $\ell_i + \overline{s}_{i,i} \geq 1$ and (7b) is satisfied and disappears.

Once clauses (7a)–(7d) have been generated for all circuit components, we obtain a greater-than-or-equal-to-k constraint by adding the unit clause $s_{n,k} \geq 1$. Analogously, a less-than-or-equal-to-k constraint is enforced using the clause $\overline{s}_{n,k+1} \geq 1$. A common optimization, known as *k-simplification*, is to omit clauses corresponding to the computation of variables $s_{i,j}$ for $j > k + 1$, as such variables are not relevant for deciding whether the cardinality constraint is true or not.

As a preparation for our proof logging discussions, let us study the variables $s_{i,j}$ in more detail, ignoring k-simplification for now. Since $s_{i,j}$ is true if and only if $\sum_{t=1}^{i} \ell_t \geq j$ holds, for all $i \in [n]$ we should be able to deduce

$$\textstyle\sum_{t=1}^{i} \ell_t = \sum_{j=1}^{i} s_{i,j} \, . \tag{8}$$

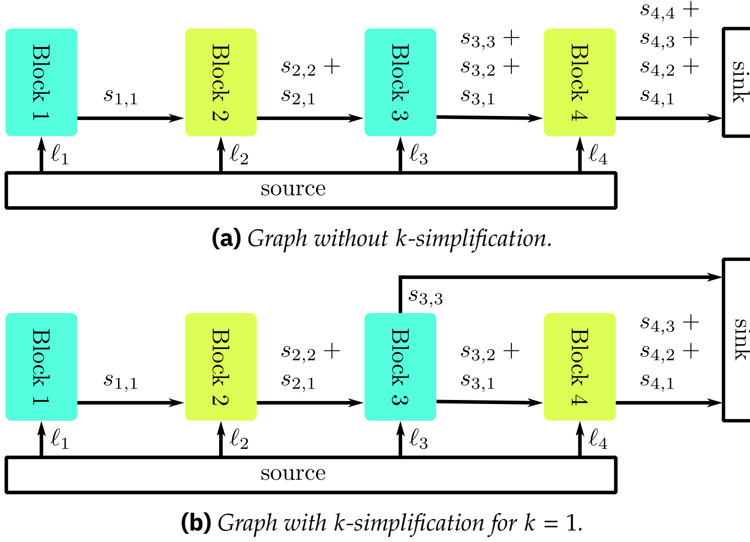
However, the sequential counter circuit computes the variables $s_{i,j}$ in the ith block using only the variables $s_{i-1,j}$ from the previous block and the literal ℓ_i, and so if we only reason locally about the ith block what we can derive is the equality

$$\ell_i + \textstyle\sum_{j=1}^{i-1} s_{i-1,j} = \sum_{j=1}^{i} s_{i,j} \, . \tag{9}$$

If we look at the variables on wires entering and exiting the ith block of the circuit, we see that Equation (9) specifies that the sum of the inputs is equal to the sum of the outputs. If we represent the circuit in Figure 2b as a graph with every block contracted into a single node and the literals ℓ_i in the cardinality constraint collected into another separate node, then every ith block node has an incoming edge from the literals node and (for $i > 1$) another edge from the $(i-1)$th block node, and an outgoing edge to the $(i+1)$th block node (or, for $i = n$, to a special sink node that we can also introduce). If we label the incoming edges by ℓ_i and $\sum_{j=1}^{i-1} s_{i-1,j}$ and the outgoing edge by $\sum_{j=1}^{i} s_{i,j}$, as shown in Figure 3a, then we can view (9) as saying that for all vertices in the graph the sum of the labels of input edges should be equal to the sum of the output edge. We will refer to this as a *preservation equality*. What is not at all obvious from this particular example, but what we will show in later sections, is that many CNF translations of pseudo-Boolean constraints can be represented as graphs with preservation equalities in a similar way, though sometimes with larger coefficients in the linear combinations of the literals. And, jumping ahead a bit, our main contribution in this paper is a generic proof logging method that will certify correctness for any CNF encoding that can be represented in this graph framework with preservation equalities.

Using the graph representation we can easily see that the telescoping sum of the preservation equalities for all nodes derives (8). From this, in turn, it is clear that a constraint on the input variables $\sum_{j=1}^{n} \ell_i \bowtie k$ implies the same constraint on the output variables, and formally this can be obtained by one final telescoping sum step combining $\sum_{j=1}^{n} \ell_i \bowtie k$ and $\sum_{j=1}^{n} \ell_i = \sum_{j=1}^{n} s_{n,j}$ to get

$$\textstyle\sum_{j=1}^{n} s_{n,j} \bowtie k \, . \tag{10}$$



(a) *Graph without k-simplification.*

(b) *Graph with k-simplification for* $k = 1$.

Figure 3: *Graph representation of the sequential counter encoding.*

Another important property of the variables $s_{i,j}$ is that they do not just take any values satisfying (9), but are ordered—since $s_{i,j}$ encodes $\sum_{t=1}^{i} \ell_t \geq j$, it follows that $s_{i,j}$ cannot be true unless also $s_{i,j'}$ is true for all $j' < j$. This can be expressed by *ordering constraints*

$$s_{i,j} \geq s_{i,j+1} \qquad i \in [n]\,, j \in [i-1]\;, \tag{11}$$

which are semantically implied by the circuit encoding.

Taking this view of the circuit encoding, the task of certifying the correctness of the CNF translation becomes surprisingly simple. If we can derive the pseudo-Boolean constraints (9)–(11), then it can be verified that the clauses of the sequential counter encoding (i.e., (7a)–(7d) plus $\bar{s}_{n,k+1} \geq 1$ and/or $s_{n,k} \geq 1$) all follow by reverse unit propagation. This is so since when asserting the clauses to false, the ordering constraints (11) will propagate enough variables $s_{i,j}$ for (9) to be falsified.

To see how to obtain the constraints (9)–(11), note that we already discussed above how to derive (10) by a telescoping sum over constraints (9), which is straightforward to do with standard cutting planes rules. To get constraints on the form (9), we can use reification to define the meaning of the variables $s_{i,j}$ by constraints

$$s_{i,j} \Leftrightarrow \ell_i + \textstyle\sum_{j=1}^{i-1} s_{i-1,j} \geq j \tag{12}$$

(with notation as introduced in (5a)–(5b) in Section 2). If we do this in increasing order for i and j, then $s_{i,j}$ is fresh in (12) and so these are valid derivation steps. From the constraints (12) we can then derive (9) and (11) as illustrated in the next example. To show the concrete syntax used in the proof file, the example is interleaved with according proof file snippets and concatenating all the snippets would result in a full proof that can be checked using the pseudo-Boolean proof checker VeriPB [EGMN20, GN21, GMN20].

Example 1. Every constraint in the proof format is assigned a unique *identifier* (ID) and constraints that are derived in the proof are annotated in this example by their corresponding identifier.

Let us consider the constraint

$$\text{(ID: 1)} \qquad x_1 + \overline{x}_2 \geq 2 \tag{13}$$

to be encoded with the sequential counter encoding. To use this constraint as input for VeriPB, the constraint is written in the OPB format [RM16], which is extended to offer, among other thing, a greater flexibility for variable names. The input file would contain the following two lines.

```
* #variables= 2 #constraints= 1
+1 x1 +1 ~x2 >= 2 ;
```

The proof file for this instance starts with the header

```
pseudo-Boolean proof version 2.0
f 1
```

to tell the checker which version of the proof format is used and to load the formula that should contain one constraint.

The proof starts with deriving the preservation equality

$$x_1 = s_{1,1} \tag{14}$$

for the first block of the sequential counter encoding. The fresh counter variable $s_{1,1}$ is introduced by reification resulting in the constraints

$$\text{(ID: 2)} \qquad \overline{s}_{1,1} + x_1 \geq 1 \tag{15a}$$

$$\text{(ID: 3)} \qquad s_{1,1} + \overline{x}_1 \geq 1 \tag{15b}$$

to be added. In the proof log the reified constraints can be added using the redundance-based strengthening rule with the according witness.

```
red +1 ~s11 +1  x1 >= 1 ; s11 -> 0
red +1  s11 +1 ~x1 >= 1 ; s11 -> 1
```

The constraints in (15) together represent the desired preservation equality (14).

For the second block the preservation equality

$$\overline{x_2} + s_{1,1} = s_{2,1} + s_{2,2} \tag{16}$$

needs to be derived. The variables $s_{2,1}$ and $s_{2,2}$ are defined by the reification constraints

$$\text{(ID: 4)} \qquad \overline{s}_{2,1} + \overline{x}_2 + s_{1,1} \geq 1 \tag{17a}$$

$$\text{(ID: 5)} \qquad 2\overline{s}_{2,2} + \overline{x}_2 + s_{1,1} \geq 2 \tag{17b}$$

$$\text{(ID: 6)} \qquad 2s_{2,1} + x_2 + \overline{s}_{1,1} \geq 2 \tag{17c}$$

$$\text{(ID: 7)} \qquad s_{2,2} + x_2 + \overline{s}_{1,1} \geq 1 \tag{17d}$$

These constraints are again introduced by redundance-based strengthening in the proof file.

```
red +1 ~s21 +1 ~x2 +1  s11 >= 1 ; s21 -> 0
red +2 ~s22 +1 ~x2 +1  s11 >= 2 ; s22 -> 0
red +2  s21 +1  x2 +1 ~s11 >= 2 ; s21 -> 1
red +1  s22 +1  x2 +1 ~s11 >= 1 ; s22 -> 1
```

This time some additional steps are required to derive the preservation equality. Adding (17a) and (17b) together yields $\bar{s}_{2,1} + 2\bar{s}_{2,2} + 2\bar{x}_2 + 2s_{1,1} \geq 3$ and dividing by 2 results in

$$(\text{ID: } 8) \qquad \bar{s}_{2,1} + \bar{s}_{2,2} + \bar{x}_2 + s_{1,1} \geq 2\,. \tag{18}$$

Adding (17c) and (17d) yields $2s_{2,1} + s_{2,2} + 2x_2 + 2\bar{s}_{1,1} \geq 3$ and dividing by 2 results in

$$(\text{ID: } 9) \qquad s_{2,1} + s_{2,2} + x_2 + \bar{s}_{1,1} \geq 2\,. \tag{19}$$

These cutting planes derivations are written to the proof log in reverse polish notation using the identifiers for the constraints. The first line derives (18) and the second line derives (19).

```
pol 4 5 + 2 d
pol 6 7 + 2 d
```

The constraints (18) and (19) together represent the desired preservation equality (16).

The next step is to sum the preservation equalities together with the input constraint (13). The sum of the constraints (15b) and (19) is $\overline{x_1} + x_2 + s_{2,1} + s_{2,2} \geq 2$ and adding (13) yields

$$(\text{ID: } 10) \qquad s_{2,1} + s_{2,2} \geq 2\,. \tag{20}$$

This cutting planes derivation is written to the proof log as the following line.

```
pol 3 9 + 1 +
```

The next step is to derive the clauses

$$(\text{ID: } 11) \qquad \bar{x}_1 + s_{1,1} \geq 1 \tag{21a}$$
$$(\text{ID: } 12) \qquad x_1 + \bar{s}_{1,1} \geq 1 \tag{21b}$$
$$(\text{ID: } 13) \qquad x_2 + s_{2,1} \geq 1 \tag{21c}$$
$$(\text{ID: } 14) \qquad \bar{s}_{1,1} + s_{2,1} \geq 1 \tag{21d}$$
$$(\text{ID: } 15) \qquad \bar{x}_2 + s_{1,1} + \bar{s}_{2,1} \geq 1 \tag{21e}$$
$$(\text{ID: } 16) \qquad x_2 + \bar{s}_{1,1} + s_{2,2} \geq 1 \tag{21f}$$
$$(\text{ID: } 17) \qquad \bar{x}_2 + \bar{s}_{2,2} \geq 1 \tag{21g}$$
$$(\text{ID: } 18) \qquad s_{1,1} + \bar{s}_{2,2} \geq 1 \tag{21h}$$

that encode the sequential counter as introduced in (7). The clauses in (21) can be derived by RUP, which is specified in the proof log by the following lines.

```
rup +1 ~x1           +1  s11 >= 1 ;
rup +1  x1           +1 ~s11 >= 1 ;
rup +1  x2           +1  s21 >= 1 ;
rup +1 ~s11          +1  s21 >= 1 ;
rup +1 ~x2   +1  s11 +1 ~s21 >= 1 ;
rup +1  x2   +1 ~s11 +1  s22 >= 1 ;
rup +1 ~x2           +1 ~s22 >= 1 ;
rup +1  s11          +1 ~s22 >= 1 ;
```

To see that these clauses follow by reverse unit propagation, we detail the RUP step for (21c) and the RUP step for the other clauses is similar. The negation of (21c) is $\overline{x}_2 + \overline{s}_{2,1} \geq 2$, which propagates x_2 and $s_{2,1}$ to false. This falsifies (17c), hence (21c) is implied.

The last step is to enforce the comparison with the degree of the constraint. As $x_1 + \overline{x}_2 \geq 2$ is satisfied if $s_{2,2}$ is true, the constraint

$$(\text{ID: } 19) \qquad s_{2,2} \geq 1 \tag{22}$$

has to be derived to enforce the comparison. This can be done using RUP, which is also written to the proof log.

```
rup +1 s22 >= 1 ;
```

This concludes our example.

To obtain the encoding with k-simplification, the most naive approach would be to simply omit the clauses enforcing correct values for the variables $s_{i,j}$ that are not used. However, this could incur a significant overhead in the proof logging when k is small, as we would always introduce $\Theta(n^2)$ intermediate variables instead of the $\Theta(kn)$ variables actually used in the final encoding. To avoid this overhead, we can introduce "overflow variables" $s_{i,k+2}$ that do not encode that the first i bits sum to $k + 2$ but instead ensure that the equality

$$\ell_i + \textstyle\sum_{j=1}^{k+1} s_{i-1,j} = \sum_{j=1}^{k+2} s_{i,j} \tag{23}$$

holds. To maintain the equality of sums over incoming and outgoing edges in our graph representation, we label the edge to the next block by $\sum_{j=1}^{k+1} s_{i,j}$ instead of $\sum_{j=1}^{i} s_{i,j}$, and introduce an additional edge going directly to the sink with the label $s_{i,k+2}$ (see Figure 3b). Note that without the additional variable $s_{i,k+2}$ we could not guarantee equality, as we would have $k + 2$ literals on the left-hand side and only $k + 1$ variables on the right-hand side.

Example 2. To apply k-simplification for $k = 1$ to Figure 3a, the output from block 3 to block 4 should only contain the sum of the two variables $s_{3,1} + s_{3,2}$. To preserve equality of the sums of inputs and outputs, we add an edge from block 3 to the sink labelled $s_{3,3}$ as in Figure 3b.

When using k-simplification, we can derive an analogue of (8) by a telescoping sum of all preservation equalities (23) yielding $\sum_{i=1}^{n} \left(\ell_i + \sum_{j=1}^{k+1} s_{i-1,j}\right) = \sum_{i=1}^{n} \left(\sum_{j=1}^{k+2} s_{i,j}\right)$, which simplifies to $\sum_{i=1}^{n} \ell_i = \sum_{i=1}^{n} s_{i,k+2} + \sum_{j=1}^{k+1} s_{n,j}$.

4 A General Framework for Certifying CNF Translations

As discussed in the introduction, there is a rich selection of encodings of pseudo-Boolean constraints in CNF. In this section, we develop a unified framework to provide proof logging for a wide range of different translations. Our approach is to represent encodings as directed graphs with preservation equalities between the incoming and outgoing edges of each node, as in our example in Figure 3, so that all clauses in the encoding can be obtained by reverse unit propagation from (telescoping sums over) these equalities. In this way, the whole proof logging task is reduced to considering a few generic ways of deriving preservation equalities. Let us start with a formal definition of the graph representation.

We will describe how the proof logging works by first introducing concrete methods that provide proof logging for different low-level steps, and then showing how these methods can be composed to certify correctness of translations from pseudo-Boolean constraints to CNF. Recall that every constraint is assigned a unique identifier. A cutting planes derivation is specified by $\mathsf{add}\,(C, D)$ to add C and D together, $\mathsf{mult}\,(C, k)$ to multiply C by k and $\mathsf{div}\,(C, k)$ to divide C by k and round up. E.g., given the previously derived constraints C and D, calling $\mathsf{add}\,(\mathsf{div}\,(C, 2)\,, \mathsf{mult}\,(D, 3))$ divides C by 2 (and rounds up), multiplies D by 3, adds the two constraints obtained in this way together, returns the resulting constraint, and writes the corresponding derivations to the proof file in reverse polish notation and using the identifiers for the constraints. A reverse unit propagation constraint C can be added using $\mathsf{rup}\,(C)$. The syntax we use for deriving a constraint by reification is $\mathsf{red}\,(z \Rightarrow C, \{z \rightarrow 0\})$ and $\mathsf{red}\,(z \Leftarrow C, \{z \rightarrow 1\})$ (where this somewhat cryptic notation is due to the fact that reification is a special case of the redundance rule in [GN21]). We use ▹ to denote comments in the pseudocode.

Definition 1 (Arithmetic Graph)**.** Let a_i, c_i be integers, ℓ_i Boolean literals, and o_i Boolean variables. An *arithmetic graph* with input $\sum_i a_i \ell_i$ and output $\sum_i c_i o_i$ is a directed multi-graph $G = (V, E)$ that satisfies the following conditions:

1. Every edge $e \in E$ has a label of the form $\sum_i b_i^e y_i^e$ for each edge $e \in E$, where b_i^e are integers and y_i^e Boolean variables.
2. There is a unique source node s that has only outgoing edges, and these edges are labelled by input literals ℓ_i in such a way that $\sum_i a_i \ell_i = \sum_{(s,v)=e\in E} \sum_i b_i^e y_i^e$.
3. There is a unique sink t that has only incoming edges, and these edges are labelled by output variables o_i in such a way that $\sum_i c_i o_i = \sum_{(v,t)=e\in E} \sum_i b_i^e y_i^e$.
4. For all other nodes v, which we refer to as *inner nodes*, the preservation equality
$$\sum_{(u,v)=e\in E} \sum_i b_i^e y_i^e = \sum_{(v,w)=e\in E} \sum_i b_i^e y_i^e \tag{24}$$
has to hold. This is saying that the sum of incoming edges equals the sum of outgoing edges, which can be derived using cutting planes with reification over the variables on outgoing edges from v.

Algorithm 6: General algorithm for translating PB constraints to CNF with proof logging.

```
1 translate_and_certify(C, f, G, F)
2     ▷ input: pseudo-Boolean constraint C of the form ∑_{i=1}^n a_i ℓ_i ⋈ k, with
        ⋈∈ {≥, ≤, =}
3     ▷ input: arithmetic graph G = (V, E) with input ∑_i a_i ℓ_i and output
        ∑_i c_i o_i
4     ▷ input: function f that takes a node and derives its preservation
        equality
5     ▷ input: set of clauses F with CNF encoding to be derived
6     sum constraints f(v) for v ∈ V in topological order to obtain
        ∑_i a_i ℓ_i = ∑_i c_i o_i;
7     combine ∑_i a_i ℓ_i = ∑_i c_i o_i and C to obtain ∑_i c_i o_i ⋈ k;
8     derive each clause in the CNF encoding F with reverse unit
        propagation (RUP);
```

The arithmetic graph does not necessarily have to be acyclic, but an acyclic graph simplifies the arguments for correctness of the generated proof.

The rest of this section will be devoted to discussing how preservation equalities (24) can be derived for different types of pseudo-Boolean expressions. Before doing so, let us just note for the record that if we have an arithmetic graph for an encoding of a pseudo-Boolean constraint, then by a telescoping argument as in Section 3 we can derive that the same constraint applies to the output of the graph.

Proposition 1. *Given an arithmetic graph with input* $\sum_i a_i \ell_i$ *and output* $\sum_i c_i o_i$ *and a PB constraint* $\sum_i a_i \ell_i \bowtie k$ *for* $\bowtie \in \{\geq, \leq, =\}$*, we can derive* $\sum_i c_i o_i \bowtie k$ *using cutting planes.*

Proof. By item 4 in Definition 1, we can derive preservation equalities (24) for all inner nodes in the graph. By summing the preservation equalities for all inner nodes together (i.e., adding up separately all greater-than-or-equal constraints and all less-than-or-equal constraints, as explained in Section 2), we obtain $\sum_i a_i \ell_i = \sum_i c_i o_i$, and combining this with $\sum_i a_i \ell_i \bowtie k$ yields $\sum_i c_i o_i \bowtie k$ as desired. □

Once the bound on the input literals is translated to a bound on the output variables, all clauses of the CNF encoding will follow by reverse unit propagation. This results in the general proof logging method shown in Algorithm 6. Note that the nodes of the graph should be traversed in topological order when deriving the preservation equalities—this is so that the variables used in the reification steps are all fresh.

Let us now discuss three different ways of representing values of natural numbers that are used in preservation equality for inner nodes. Perhaps the most straightforward way to encode a number j with domain $A = \{0, 1, \ldots, m\} \subseteq \mathbb{N}_0$ with Boolean variables is to write j in unary with variables z_i so that $j = \sum_{i \in [m]} z_i$. In such an encoding we can also require, using constraints $z_i \geq z_{i+1}$, that the variables z_i are ordered so that z_i is true if and only if $j \geq i$. This means that

Algorithm 7: Deriving a unary sum over fresh variables z_i.

```
1  derive_unary_sum(C')
2      ▹ input: C' of the form Σ_{i=1}^n ℓ_i = Σ_{i=1}^n z_i and describing the constraint
         to be derived
3      ▹ the z_i variables need to be fresh, the left-hand side is the sum to be
         encoded
4      for j = 1, ..., k do
5          ▹ Step 7.1: introduce variables
6          D_j^⇒, D_j^⇐ ← reify(z_j ⇔ Σ_{i=1}^n 1 · ℓ_i ≥ j);
7      ▹ Step 7.2: derive Σ_{i=1}^n ℓ_i ≥ Σ_{i=1}^n z_i
8      C^⇒ ← derive_sum(D_1^⇒, D_2^⇒, ..., D_n^⇒);
9      ▹ Step 7.3: derive Σ_{i=1}^n ℓ_i ≤ Σ_{i=1}^n z_i
10     C^⇐ ← derive_sum(D_n^⇐, D_{n-1}^⇐, ..., D_1^⇐);
11     for i = 1, ..., k − 1 do
12         ▹ Step 7.4: derive z_i ≥ z_{i+1}, i ∈ [n − 1]
13         derive_ordering(D_i^⇐, D_{i+1}^⇒);
14     return C^⇒, C^⇐;
```

Algorithm 8: Reify $\sum_{i=1}^n a_i \ell_i \geq j$ using the fresh variable z_j.

```
1  reify(z_j ⇔ Σ_{i=1}^n a_i ℓ_i ≥ j)
2      ▹ z_j ⇒ Σ_{i=1}^n a_i ℓ_i ≥ j
3      C^⇒ ← red(Σ_{i=1}^n a_i ℓ_i + j z̄_j ≥ j, {z_j → 0});
4      ▹ z_j ⇐ Σ_{i=1}^n a_i ℓ_i ≥ j
5      C^⇐ ← red(Σ_{i=1}^n a_i ℓ̄_i + (Σ_{i=1}^n a_i − j + 1) z_j ≥ Σ_{i=1}^n a_i − j + 1, {z_j → 1});
6      return C^⇒, C^⇐;
```

listing the variables in reverse order $z_m, z_{m-1}, \ldots, z_1$ yields the number j written in unary (after a prefix of zeros). This is known as the *order encoding*, and this type of representation is used in the sequential counter [Sin05] and totalizer [BB03] encodings. We can certify the correctness of this encoding as stated in the next proposition.

Proposition 2 (Unary Sum)**.** *For literals ℓ_i and fresh variables z_i, $i \in [n]$, the constraints*

$$\sum_{i=1}^n \ell_i \geq \sum_{i=1}^n z_i \tag{25a}$$

$$\sum_{i=1}^n \ell_i \leq \sum_{i=1}^n z_i \tag{25b}$$

$$z_i \geq z_{i+1} \qquad i \in [n-1] \tag{25c}$$

can be derived in $\mathrm{O}(n)$ *steps in cutting planes with reification. Thus, the variable z_i is defined to be true if and only if at least i literals are true.*

Algorithm 9: Derive sum of reification variables.

```
1 derive_sum(D_1, ..., D_n)
2     ▹ input: D_j is of the form ∑_{i=1}^n ℓ_i + j z̄_j ≥ j
3     C ← 0 ≥ 0;
4     for j from 1 to n do
5         C ← div (add (mult (C, j − 1), D_j), j);
6         ▹ Invariant: C : ∑_{i=1}^n ℓ_i + ∑_{i=1}^j z̄_i ≥ j
7     return C;
```

Algorithm 10: Deriving an ordering constraint $z_A \geq z_B$ from the reification constraints.

```
1 derive_ordering(C, D)
2     ▹ input: C is of the form z_A ⇒ ∑_{i=1}^n a_i ℓ_i ≥ A
3     ▹ input: D is of the form z_B ⇐ ∑_{i=1}^n a_i ℓ_i ≥ B
4     divisor ← ∑_{i=1}^n a_i;
5     ▹ derive z_A ≥ z_B if A < B
6     div (add (C, D), divisor);
```

Proof. The unary sum constraints in (25) can be derived using Algorithm 7. We will show the correctness of Algorithm 7 first and then that the derivation following this algorithm requires $O(n)$ steps in cutting planes with reification.

Algorithm 7 is split into four major steps. Step 7.1 is to introduce the fresh variables z_j as reifications of the constraints $\sum_{i=1}^n \ell_i \geq j$, which is shown in Algorithm 8 for the more general case of arbitrary positive coefficients.

In Step 7.2 the lower bound (25a) is derived using Algorithm 9 maintaining the invariant $\sum_{i=1}^n \ell_i + \sum_{i=1}^j \bar{z}_i \geq j$ after each iteration. For the base case $j = 1$, the invariant is equivalent to the reification constraint $z_1 \Rightarrow \sum_{i=1}^n \ell_i \geq 1$, which in normalized form is $\sum_{i=1}^n \ell_i + \bar{z}_1 \geq 1$ and hence this case is covered. For the inductive step, to go from $j-1$ to j we multiply the invariant $\sum_{i=1}^n \ell_i + \sum_{i=1}^{j-1} \bar{z}_j \geq j-1$ by $j-1$ and add the reification constraint $z_j \Rightarrow \sum_{i=1}^n \ell_i \geq j$, which is $\sum_{i=1}^n \ell_i + j\bar{z}_j \geq j$ in normalized form, to get $j \sum_{i=1}^n \ell_i + (j-1) \sum_{i=1}^{j-1} \bar{z}_i + j\bar{z}_j \geq j^2 - j + 1$. Division by j and rounding up yields $\sum_{i=1}^n \ell_i + \sum_{i=1}^j \bar{z}_i + \bar{z}_j \geq j$, i.e., the invariant for j. For $j = n$ the invariant is the normalized form of (25a).

In Step 7.3 the upper bound (25b) is again derived using Algorithm 9, except that the constraints are processed in reverse order (just as in Example 1 on page 81).

In Step 7.4 the ordering constraint is derived using Algorithm 10, using the reification constraints. Algorithm 10 handles the general case of deriving the ordering constraint $z_j \geq z_{j+1}$ from any reification constraints $z_{j+1} \Rightarrow \sum_{i=1}^n a_i \ell_i \geq j+1$ and $z_j \Leftarrow \sum_{i=1}^n a_i \ell_i \geq j$. For the sequential counter encoding the coefficients a_i are all 1. In normalized form these two constraints are $(j+1)\bar{z}_{j+1} + \sum_{i=1}^n a_i \ell_i \geq j+1$ and $(m-j+1)z_j + \sum_{i=1}^n a_i \bar{\ell}_i \geq m-j+1$, where $m = \sum_{i=1}^n a_i$. Adding both constraints

together yields $(m - j + 1)z_j + (j + 1)\overline{z}_{j+1} \geq 2$ and we get the desired ordering constraint after division by a large enough number, such as m.

Now that the correctness of Algorithm 7 is established, all that remains is to verify that this algorithm uses $\mathrm{O}(n)$ steps in cutting planes with reification. Step 7.1 uses n reification steps, Step 7.2 and 7.3 uses $3(n-1)$ cutting planes steps each and Step 7.4 uses $2(n-1)$ cutting planes steps in the worst case. Thus, in total $\mathrm{O}(n)$ steps in cutting planes with reification are used. □

A concrete illustration of how these derivations can be done was given in Example 1 (with ℓ_3, $s_{2,1}$, and $s_{2,2}$ playing the roles of the literals ℓ_i and $s_{3,j}$, $j \in [3]$, being the fresh variables).

When encoding the value of a number j that can only take a small number of values in a large range, it is wasteful to introduce variables for all values in the range. For example, if $j \in \{0, 50, 75\}$, then the first 50 variables in a full unary representation are either all true or all false, but will never take different values. In such cases we can instead use what we will refer to as a *sparse unary encoding*, where in our example $j \in \{0, 50, 75\}$ would be represented as $50 \cdot z_{50} + 25 \cdot z_{75}$, where we enforce $z_{50} \geq z_{75}$. More formally, for a (finite) domain $A \subseteq \mathbb{N}_0$ and variables $\vec{z} = \{z_i \mid i \in A \cup \{\infty\}\}$ we define

$$\mathsf{sparse}(\vec{z}, A) \doteq \textstyle\sum_{i \in A \setminus \{0\}} (i - \mathsf{pred}(i, A)) \cdot z_i \,, \tag{26a}$$

where $\mathsf{pred}(i, A) = \max\{j \in A \cup \{0\} \mid j < i\}$, and we also use constraints

$$z_i \geq z_{\mathsf{succ}(i,A)} \qquad\qquad i \in A \setminus \{\max(A)\} \tag{26b}$$

to enforce that the variables z_i are ordered, where $\mathsf{succ}(i, A) = \min\{j \in A \cup \{\infty\} \mid j > i\}$ is the successor of i in A. This representation is used in the sequential weight counter [HMS12] and generalized totalizer [JMM15] encodings, and we can certify correctness for it as stated next.

Proposition 3 (Sparse Unary Sum)**.** *Let $A, B \subseteq \mathbb{N}_0$ be given with sparse encodings* $\mathsf{sparse}(\vec{x}, A)$ *and* $\mathsf{sparse}(\vec{y}, B)$ *as in* (26a)–(26b). *Then for $E = \{i + j \mid i \in A, j \in B\}$ and fresh variables $\vec{z}$ we can derive*

$$\mathsf{sparse}(\vec{x}, A) + \mathsf{sparse}(\vec{y}, B) = \mathsf{sparse}(\vec{z}, E) \tag{27a}$$

$$z_i \geq z_{\mathsf{succ}(i,E)} \qquad\qquad i \in E \setminus \{\max(E)\} \tag{27b}$$

in cutting planes with reification using $\mathrm{O}(|A| \cdot |B|)$ *steps.*

Proof. The proposition is proven by presenting and analyzing Algorithm 11, which given two numbers in sparse unary representation derives their sum. Just as for the unary sum, we start in Step 11.1 by introducing the required fresh variables via reification. However, we only need to introduce the variables with index in E. If k-simplification is used, then also variables with index bigger than k need to be introduced, as without them equality cannot be derived. The ordering constraints can be derived as before using Algorithm 10.

In Step 11.2 we introduce a variable z_{eq} which is true if and only if the equality to be derived is true. Since an equality is actually two inequalities, we need to

Algorithm 11: Deriving a sparse unary sum over fresh variables $\vec{z}$.

```
1  derive_sparse_unary_sum(C′)
2      ▷ input: C′ of the form sparse(x⃗, A) + sparse(y⃗, B) = sparse(z⃗, E) and
         describing the constraint to be derived such that A, B ⊆ ℕ,
         E = {i + j | i ∈ A, j ∈ B}
3      ▷ Step 11.1: Introduce variables as reification and derive ordering
4      for j ∈ E \ {0} do
5          D_j^⇒, D_j^⇐ ← reify(z_j ⇔ sparse(x⃗, A) + sparse(y⃗, B) ≥ j);
6      for i ∈ E \ {0, max(E)} do
7          derive_ordering ( D_i^⇐, D_succ(i,E)^⇒ ) ;        ▷ derive z_i ≥ z_succ(i,E)
8      ▷ Step 11.2: : reify constraint to be derived
9      C^⇒, _ ← reify(z_geq ⇔ sparse(x⃗, A) + sparse(y⃗, B) ≥ sparse(z⃗, E));
10     C^⇐, _ ← reify(z_leq ⇔ sparse(x⃗, A) + sparse(y⃗, B) ≤ sparse(z⃗, E));
11     reify(z_eq ⇔ z_geq + z_leq ≥ 2);
12     ▷ Step 11.3: derive that z_eq ≥ 1
13     try_all_values(sparse(x⃗, A), sparse(y⃗, B), z_eq);
14     ▷ Step 11.4: derive constraint to be derived from its reification
15     M ← max(E) ; ▷ choose M equal to coefficient of reification variables
16     D ← rup (z_geq ≥ 1);
17     C^⇒ ← add (C^⇒, mult (D, M));
18     D ← rup (z_leq ≥ 1);
19     C^⇐ ← add (C^⇐, mult (D, M));
20     return C^⇒, C^⇐;
```

introduce separate variables z_{geq}, z_{leq} for each inequality and then combine them into z_{eq}.

In Step 11.3 we derive $z_{eq} \geq 1$ by checking all combinations of values in A and B, which requires $O(|A| \cdot |B|)$ steps.

In Step 11.4 we use that $z_{eq} \geq 1$ and hence $z_{geq} = z_{leq} = 1$, which allows us to derive $\text{sparse}(\vec{x}, A) + \text{sparse}(\vec{y}, B) \geq \text{sparse}(\vec{z}, E)$ and $\text{sparse}(\vec{x}, A) + \text{sparse}(\vec{y}, B) \leq \text{sparse}(\vec{z}, E)$ by removing z_{geq} and z_{leq} from the constraints introduced in Step 11.2.

Since Step 11.1 and 11.3 require $O(|A| \cdot |B|)$ steps each and the number of steps for Step 11.2 and 11.4 is in $O(1)$, the total number of cutting planes with reification steps is $O(|A| \cdot |B|)$. Asymptotically, this is the same number of steps required to compute which elements are in E, so this is still linear in the time needed to construct the encoding. □

As in the case of the unary sum in Proposition 2, adding the constraints (27a)–(27b) maintains equisatisfiability, because the fresh variables $\vec{z}$ are free to take values so that the constraints are satisfied. The general idea is again to introduce $\vec{z}$ via reification, but the rest of the proof of Proposition 3 gets a bit more complicated—we have to perform a brute-force search on the possible combinations of values for A and B, showing that the equality holds in all cases, and provide a proof log for the correctness of this backtracking search.

Algorithm 12: Given a reified sparse unary sum, derive that the reification variable is true.

```
1  ▹ helper function:
2  fix(sparse(x⃗, A), a)
3  |  return x̄_a + x_succ(a,A) ;                    ▹ replace x_0 by 1 and x_∞ by 0
4  ▹ main function:
5  try_all_values(sparse(x⃗, A), sparse(y⃗, B), z_eq)
6  |  C_outer ← rup (0 ≥ 0);
7  |  for i ∈ A do
8  |  |  C_inner ← rup (0 ≥ 0);
9  |  |  for j ∈ B do
10 |  |  |  ▹ a (respectively b) is the value encoded by sparse(x⃗, A)
   |  |  |    (sparse(y⃗, B))
11 |  |  |  ▹ encode that (a = i ∧ b = j) ⇒ z_eq
12 |  |  |  D ← rup (fix(sparse(x⃗, A), i)
13 |  |  |            + fix(sparse(y⃗, B), j) + z_eq ≥ 1) ;
14 |  |  |  C_inner ← add (C_inner, D);
15 |  |  C_outer ← add (C_outer, div (C_inner, |B|));
16 |  C_outer ← div (C_outer, |A|);
17 |  return C_outer ;                                ▹ C_outer is now z_eq ≥ 1
```

To illustrate how the derivation in Algorithm 11 works, let us consider an example.

Example 3. Let the set of possible values for the left child node be $A = \{0, 2\}$ and the corresponding set for the right child node be $B = \{0, 2, 4\}$. Hence, the set of possible output values is $E = \{0, 2, 4, 6\}$. Step 11.1 derives the reified constraints

$$z_2 \Leftrightarrow 2x_2 + 2y_2 + 2y_4 \geq 2 \tag{28a}$$
$$z_4 \Leftrightarrow 2x_2 + 2y_2 + 2y_4 \geq 4 \tag{28b}$$
$$z_6 \Leftrightarrow 2x_2 + 2y_2 + 2y_4 \geq 6 \tag{28c}$$

and the ordering constraints $z_2 \geq z_4$ and $z_4 \geq z_6$.

Then Step 11.2 uses reification to derive the constraints

$$6\overline{z}_{geq} + 2x_2 + 2y_2 + 2y_4 + 2\overline{z}_2 + 2\overline{z}_4 + 2\overline{z}_6 \geq 6 \tag{29a}$$
$$6\overline{z}_{leq} + 2\overline{x}_2 + 2\overline{y}_2 + 2\overline{y}_4 + 2z_2 + 2z_4 + 2z_6 \geq 6 \tag{29b}$$
$$z_{eq} \Leftrightarrow z_{geq} + z_{leq} \geq 2\,. \tag{29c}$$

Then Step 11.3 derives $z_{eq} \geq 1$ using Algorithm 12 by checking all combinations of values in A and B. After the first iteration of the outer loop in Algorithm 12 the clauses

$$x_2 + y_2 + z_{eq} \geq 1 \tag{30a}$$
$$x_2 + \overline{y}_2 + y_4 + z_{eq} \geq 1 \tag{30b}$$
$$x_2 + \overline{y}_4 + z_{eq} \geq 1 \tag{30c}$$

have been derived. Deriving (30a) by RUP sets $x_2 = y_2 = z_{eq} = 0$. This causes the ordering constraints to propagate all variables in $\vec{x}$ and $\vec{y}$. As all $\vec{x}$ and $\vec{y}$ variables are set, the reification constraints introduced in Step 11.1 will cause all $\vec{z}$ variables to propagate. As the constraints reified in Step 11.2 are satisfied, $z_{geq} = z_{leq} = 1$ is propagated and hence z_{eq} should be 1. However, RUP already set z_{eq} to 0, which is a contradiction showing that (30a) can be derived. Deriving the other clauses works analogously. Adding all clauses in (30) together results in $3x_2 + 3z_{eq} \geq 1$, which is divided by 3 to obtain

$$x_2 + z_{eq} \geq 1\,. \tag{31}$$

Analogously, in the second iteration we derive the constraints

$$\overline{x}_2 + y_2 + \phantom{\overline{y}_2 + y_4 +} z_{eq} \geq 1 \tag{32a}$$

$$\overline{x}_2 + \overline{y}_2 + y_4 + z_{eq} \geq 1 \tag{32b}$$

$$\overline{x}_2 + \phantom{\overline{y}_2 +} \overline{y}_4 + z_{eq} \geq 1 \tag{32c}$$

using RUP and then

$$\overline{x}_2 + z_{eq} \geq 1 \tag{33}$$

by adding all the constraints in (32) together and dividing the result by 3. Adding the constraints (31) and (33) together yields $2z_{eq} \geq 1$ and dividing by 2 results in $z_{eq} \geq 1$.

Step 11.4 first computes the coefficient of z_{geq} and z_{leq}, which is $M = 6$. Then the constraints $z_{geq} \geq 1$ and $z_{leq} \geq 1$ are derived using RUP by setting either $z_{geq} = 0$ or $z_{leq} = 0$. Then $z_{eq} \geq 1$ propagates $z_{eq} = 1$. However, (29c) propagates $z_{eq} = 0$, which is a contradiction. Then $z_{geq} \geq 1$ and $z_{leq} \geq 1$ are multiplied by 6 and added to (29a) and (29b), respectively. This yields constraints

$$2x_2 + 2y_2 + 2y_4 + 2\overline{z}_2 + 2\overline{z}_4 + 2\overline{z}_6 \geq 6 \tag{34a}$$

$$2\overline{x}_2 + 2\overline{y}_2 + 2\overline{y}_4 + 2z_2 + 2z_4 + 2z_6 \geq 6\,, \tag{34b}$$

which together represent the preservation equality for the sparse unary sum.

If we perform sums repeatedly as in Proposition 3, then the size of the domain can double in every step in the worst case, leading to an exponential explosion (this happens, for instance, if all values in the domains are distinct powers of 2). The third encoding we consider addresses this worst-case scenario by using a *binary encoding* $j = \sum_{i=0}^{\lfloor \log_2(m) \rfloor} 2^i \cdot z_i$. To compute the binary representation, it is sufficient—as we will discuss next in Section 5—to compose multiple full adders, which compute the sum of up to three input bits, using a binary adder circuit as described in [ES06].

Proposition 4. *For literals ℓ_1, ℓ_2, ℓ_3 and fresh variables c, s, we can derive the equality*

$$\ell_1 + \ell_2 + \ell_3 = 2c + s \tag{35}$$

in cutting planes with reification using O(1) *steps.*

Algorithm 13: Proof logging for the encoding of a single full adder.

```
1 full_adder(x,y,z)
2   D_carry^⇒, D_carry^⇐ ← reify(c ⇔ x + y + z ≥ 2);
3   D_sum^⇒, D_sum^⇐ ← reify(s ⇔ x + y + z + 2c̄ ≥ 3);
4   D^⇒ ← div(add(mult(D_carry^⇒, 2), D_sum^⇒), 3);
5   D^⇐ ← div(add(mult(D_carry^⇐, 2), D_sum^⇐), 3);
6   ▹ D is the preservation equality of the full adder
7   return D^⇒, D^⇐, c, s;
```

Proof. Algorithm 13 can be used to derive the constraints that represent the preservation equality (35) for a single binary full adder.

Algorithm 13 first derives

$$c \Leftrightarrow \ell_1 + \ell_2 + \ell_3 \geq 2 \tag{36a}$$

$$s \Leftrightarrow \ell_1 + \ell_2 + \ell_3 + 2\overline{c} \geq 3 \tag{36b}$$

using reification, since c and s are fresh variables, and then multiplies (36a) by 2, add (36b), and divides the result by 3. To show how this works for the $\Rightarrow$-direction of the reification, 2 times (36a) is $4\overline{c} + 2\ell_1 + 2\ell_2 + 2\ell_3 \geq 4$, adding $3\overline{s} + \ell_1 + \ell_2 + \ell_3 + 2\overline{c} \geq 3$ as in (36b) yields $6\overline{c} + 3\overline{s} + 3\ell_1 + 3\ell_2 + 3\ell_3 \geq 7$, and dividing by 3 gives us $2\overline{c} + \overline{s} + \ell_1 + \ell_2 + \ell_3 \geq 3$ as desired. The other direction is equivalent. We refer the reader to [GN21] for more details.

This algorithm uses 2 reification steps and 6 cutting planes steps. Thus, the number of cutting planes with reification steps is in O(1). □

Again, it should be clear that this maintains equisatisfiability, since the carry-out bit c and sum bit s can be set appropriately.

5 Certifying the Binary Adder Network Encoding

Now that this general framework has been introduced, we show how it can be applied to implement proof logging for some specific pseudo-Boolean to CNF encodings. In this section, we will consider the so-called *binary adder encoding* [ES06].

The idea behind the binary adder encoding is to use an adder network to compute the value of $\sum_i a_i\ell_i$ as a binary number $\sum_{i=0}^{bits} 2^i o_i$, where $bits = \lfloor \log_2(\sum_i a_i) \rfloor$ is the required *bit width*, and then compare this to the right-hand side constant in the constraint $\sum_i a_i\ell_i \bowtie k$.

To recapitulate the algorithm for adder network construction in [ES06], let us say that a 2^m*-bit* is a literal representing the numerical value 2^m and that a 2^m*-bucket* is a queue of 2^m-bits. We use $[m]_2$ to denote the binary representation of a natural number m. The algorithm starts by initializing each 2^m-bucket with all literals ℓ_i in $\sum_i a_i\ell_i \bowtie k$ such that the 2^m-bit of $\lfloor a_i \rfloor_2$ is 1. Then for m in increasing

Algorithm 14: Construction of adder network [ES06]. Procedure full_adder adds full adder to network.

```
1  adder_network(b)
2      ▹ input: vector of buckets b
3      for i from 0 to b.size() do
4          while b_i.size() ≥ 2 do
5              if b_i.size() = 2 then
6                  (x, y) ← b_i.dequeue();
7                  (c, s) ← full_adder(x, y, 0);
8              else
9                  (x, y, z) ← b_i.dequeue();
10                 (c, s) ← full_adder(x, y, z);
11             b_i.enqueue(s);
12             b_{i+1}.enqueue(c);
```

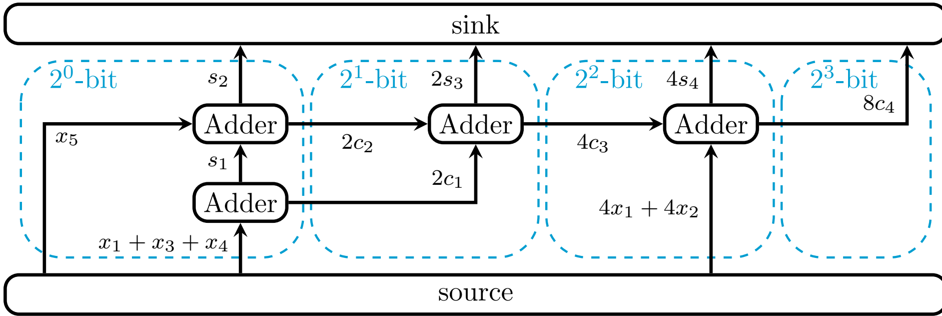


Figure 4: *Layout of arithmetic graph for adder network encoding of* $5x_1+4x_2+x_3+x_4+x_5$.

order we repeat the following procedure: while there are at least 2 elements in the 2^m-bucket, dequeue three bits x, y, z, or set $z = 0$ if there are exactly 2 bits left. Use x, y, and z as input for a new full adder with fresh variables c and s as output (these are just placeholder names), and insert s in the 2^m-bucket and c in the 2^{m+1}-bucket (possibly creating a new bucket). See Algorithm 14 for the pseudocode to generate this encoding.

The arithmetic graph is obtained from the adder network by representing each full adder by a node. Each inner node constructed from a 2^m-bucket has 3 input edges with labels $2^m \cdot x$, $2^m \cdot y$, and $2^m \cdot z$ and 2 output edges with labels $2^m \cdot s$ and $2^{m+1} \cdot c$. An example for the PB expression $5x_1+4x_2+x_3+x_4+x_5$ is shown in Figure 4. The preservation equality can be derived using Proposition 4 and multiplying the resulting equality $x+y+z = 2c+s$ by 2^m to obtain $2^m \cdot x+2^m \cdot y+2^m \cdot z = 2^{m+1} \cdot c+2^m \cdot s$. When the construction algorithm ends, each 2^m-bucket has at most one 2^m-bit left, and we connect the corresponding edges to the sink, resulting in an output of the form $\sum_{i=0}^{bits} 2^i \cdot o_i$. If the 2^i-bucket is empty, o_i is fixed to 0.

Each full adder of the network is encoded to CNF using clauses

$$\begin{array}{llll} & \overline{x}+\overline{y}+\overline{z}+s \geq 1 & & x+y+z+\overline{s} \geq 1 \\ \overline{y}+\overline{z}+c \geq 1 & \overline{x}+y+z+s \geq 1 & y+z+\overline{c} \geq 1 & x+\overline{y}+\overline{z}+\overline{s} \geq 1 \\ \overline{x}+\overline{z}+c \geq 1 & x+\overline{y}+z+s \geq 1 & x+z+\overline{c} \geq 1 & \overline{x}+y+\overline{z}+\overline{s} \geq 1 \\ \overline{x}+\overline{y}+c \geq 1 & x+y+\overline{z}+s \geq 1 & x+y+\overline{c} \geq 1 & \overline{x}+\overline{y}+z+\overline{s} \geq 1 \end{array} \tag{37}$$

which are all RUP with respect to the preservation equality $x + y + z = 2c + s$.

To compare the constant k in the PB constraint with the output of the circuit, we encode a bitwise comparison $\vec{x} \geq \vec{y}$ for bit vectors $\vec{x}$ and $\vec{y}$, where $\vec{x} = o_{bits} \cdots o_1 o_0$ and $\vec{y} = [k]_2$ or vice versa, depending on whether we want to encode $\sum_{i=1}^{n} a_i \ell_i \geq k$ or $\sum_{i=1}^{n} a_i \ell_i \leq k$, respectively. The following encoding is standard and can also be found in [ES06]. For $\sum_{i=1}^{n} a_i \ell_i = k$, comparisons for both directions are performed. If the sizes of the two vectors are different, the shorter vector is padded with 0, after which the constraints

$$x_i + \overline{y}_i + \textstyle\sum_{j=i+1}^{bits} x_j \overline{y}_j + \overline{x}_j y_j \geq 1 \qquad i = 0, 1, \ldots, bits \tag{38}$$

are added to the CNF encoding. Since either $\vec{x}$ or $\vec{y}$ is a vector of constant bits, the constraints (38) are indeed clauses. Basically, the encoding compares the two numbers from the most-significant bit to the least-significant bit. It is only required to check the biggest first index i, where x_i and y_i are different. Then the corresponding clause (38) for index i is only satisfied if $x_i \geq y_i$. The clauses (38) are RUP with respect to the constraint $\sum_{i=0}^{bits} 2^i \cdot o_i \bowtie k$, which we obtain from the arithmetic graph using Proposition 1. To see this, note that asserting (38) to false will set all 2^j-bits for $j > i$ equal but the 2^i-bits to opposite values, which immediately falsifies $\sum_{i=0}^{bits} 2^i \cdot o_i \bowtie k$.

6 Certifying the Totalizer and Generalized Totalizer Encodings

To show that the framework developed in Section 4 can be applied to many different pseudo-Boolean to CNF encodings, we detail in this section how our framework can be applied to add certification to the totalizer [BB03] and generalized totalizer [JMM15] encoding.

The totalizer and generalized totalizer encodings accumulate the input in the form of a balanced binary tree. The totalizer encodes cardinality constraints and uses the order encoding to represent values, while the generalized totalizer encodes general pseudo-Boolean constraints and uses a sparse representation. An example of an arithmetic graph for the generalized totalizer encoding is shown in Figure 5. The nodes are combined in form of a binary tree, where we ensure that the value is preserved for each inner node. To perform k-simplification, the arithmetic graph has additional edges that go directly to the sink node. The formal definition of the arithmetic graph for the (generalized) totalizer encoding is as follows.

Definition 2 (Arithmetic graph for the generalized totalizer encoding)**.** Given a linear sum $\sum_i a_i \ell_i$ over n variables, let G be a binary tree with edges directed

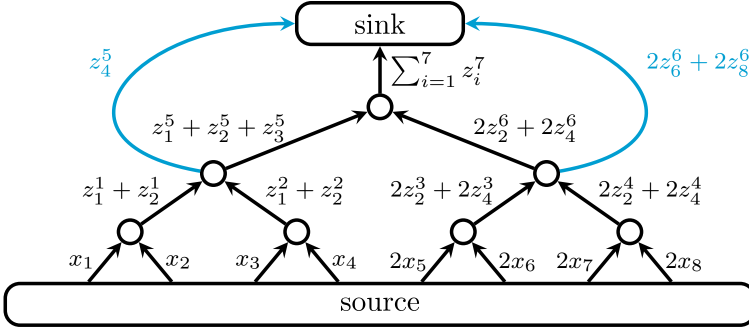


Figure 5: *Layout of the arithmetic graph for the generalized totalizer encoding of* $x_1 + x_2 + x_3 + x_4 + 2x_5 + 2x_6 + 2x_7 + 2x_8 \leq 2$*. Edges introduced for k-simplification are colored cyan.*

towards the root r, leaves s_i for $i \in [n]$ and an additional sink node t with an edge (r, t). The edge (s_i, v) is labelled with $a_i \ell_i$. For an inner node v with two incoming edges labelled $\mathsf{sparse}(\vec{x}, A)$ and $\mathsf{sparse}(\vec{y}, B)$, the outgoing edge is labelled $\mathsf{sparse}(\vec{z}, E)$, where $\vec{z}$ are fresh variables and $E = \{i + j \mid i \in A, j \in B\}$. All s_i are combined into a single source node. For k-simplification we split $\mathsf{sparse}(\vec{z}, E) = \sum_{i \in E} a_i z_i$ into $\sum_{i \leq \mathsf{succ}(k,E)} a_i z_i$ and $\sum_{i > \mathsf{succ}(k,E)} a_i c_i$.

To see that this graph is an arithmetic graph, we only need to check that we can derive the preservation equality for each inner node. We can use Proposition 3 to derive the required preservation equality. Proposition 3 also requires to have ordering constraints on the input literals. However, it is easy to see by an inductive argument that the ordering constraints on the literals can also be derived as we process the nodes in topological order. For the base case, edges from source nodes only contain a single literal, which is vacuously ordered. For inner nodes we get the ordering constraints by applying Proposition 3. If E contains all integers between 0 and $\max(E)$, we can use Proposition 2 to derive the preservation equality, which requires $O(|E|)$ steps instead of $O(|A| \cdot |B|)$ steps and hence reduces overhead.

For each inner node in the graph with incoming edge labels $\mathsf{sparse}(\vec{x}, A)$ and $\mathsf{sparse}(\vec{y}, B)$, the (generalized) totalizer encoding contains the clauses

$$\overline{x}_i + \overline{y}_j + z_{i+j} \geq 1 \qquad i \in A, j \in B \tag{39a}$$

$$x_{\mathsf{succ}(i,A)} + y_{\mathsf{succ}(j,B)} + \overline{z}_{\mathsf{succ}(i+j,E)} \geq 1 \qquad i \in A, j \in B \tag{39b}$$

for $\mathsf{succ}(i, A) = \min\{j \mid j \in A \cup \{\infty\}, j > i\}$ and for x_0, y_0 replaced by 1 and $x_\infty, y_\infty, z_\infty$ by 0, with ensuing simplification.

For the proof logging of the CNF encoding we can simply add all clauses using reverse unit propagation. A RUP check of (39a) will assign $x_i = y_j = 1$ and $z_{i+j} = 0$. The ordering constraints on $\vec{x}, \vec{y}$ will propagate variables in $\vec{x}, \vec{y}$ to true so that $\mathsf{sparse}(x, A) + \mathsf{sparse}(y, B)$ has a value of at least $i + j$, while the ordering constraints on $\vec{z}$ will propagate variables in $\vec{z}$ to false so that $\mathsf{sparse}(z, E)$ can only take a value strictly less than $i + j$. This will violate the preservation equality $\mathsf{sparse}(z, E) = \mathsf{sparse}(x, A) + \mathsf{sparse}(y, B)$, showing that (39a) is indeed a RUP clause. Deriving the clause (39b) works analogously.

To enforce a pseudo-Boolean constraint $\sum_i a_i\ell_i \bowtie k$, we first derive a bound on the output of the arithmetic graph $\sum_i c_i o_i \bowtie k$, using Proposition 1. Then we can derive unit clauses on the output via reverse unit propagation.

To encode $\sum_i a_i\ell_i \geq k$ or $\sum_i a_i\ell_i \leq k$ the unit clause $z_{\mathsf{succ}(k-1,E)} \geq 1$ or $\overline{z}_{\mathsf{succ}(k,E)} \geq 1$ is added, respectively. This clause is RUP, as the derived sum $\sum_i c_i o_i$ has a value of at most $k-1$ or at least $k+1$ and thus the constraint $\sum_i c_i o_i \geq k$ or $\sum_i c_i o_i \leq k$ is falsified, respectively. To encode $\sum_i a_i\ell_i = k$ both unit clauses are added.

7 Experimental Evaluation

To evaluate the proof logging methods developed in this paper, we have implemented certified translations to CNF for the sequential counter [Sin05], adder network [ES06], totalizer [BB03], and generalized totalizer [JMM15] encodings in the tool VeritasPBLib which is publicly available at `https://github.com/forge-lab/VeritasPBLib`. This tool takes a pseudo-Boolean formula in OPB format [RM16] and returns a CNF translation with a proof logging certificate. We have employed the verifier VeriPB[1] [GN21, BGMN22] to check the certificate returned by VeritasPBLib, and have used the SAT solver Kissat[2] [BFFH20], in a lightly modified version outputting *DRAT* proofs in pseudo-Boolean format,[3] to solve the CNF formula. Finally, we have conjoined the certificates from the CNF translation and the SAT solving and verified the end-to-end pipeline with VeriPB. See [GMNO22] for source code and experimental data.

The experiments were conducted on Amazon EC2 r5.large instances (2 vCPU) with Intel(R) Xeon(R) Platinum 8259CL CPU @ 2.50GHz CPUs, 16 GB of memory, and gp2 volumes. We ran one process on each instance with a memory limit of 15 GB and a time limit of 7,200 seconds for verifying the proof with VeriPB, and a time limit of 1,800 seconds for CNF translation with VeritasPBLib and SAT solving with Kissat. We gave additional time for verification, which tends to be slower than solving the problem.

Our evaluation aimed to answer the following questions:

1. Can we use our end-to-end framework to verify the results of SAT-based pseudo-Boolean solving, and how efficient is the verification?

2. How long does the verification of the proof log take when compared to the translation of the pseudo-Boolean formula to CNF?

3. How does a verified SAT-based pseudo-Boolean approach compare against other pseudo-Boolean solvers?

4. Can we use our end-to-end framework to certify the optimal solutions of optimization problems, such as Maximum Satisfiability?

[1]VeriPB is available at `https://gitlab.com/MIAOresearch/software/VeriPB`.

[2]The original version of Kissat is available at `https://fmv.jku.at/kissat/`.

[3]Our modified version of Kissat with pseudo-Boolean proof logging is available at `https://gitlab.com/MIAOresearch/tools-and-utilities/kissat_fork`.

Table 1: *Properties of the pseudo-Boolean formulas used in the experiments.*

		Card	PB	Card+PB
	#Inst.	772	442	308
Card	Avg. #	107.01±252.57	0.00	1,154.43±5,881.78
	Avg. #Lits	36.45±47.43	0.00	16.96±26.57
	Avg. Coeff. Size	1.00±0.00	0.00	1.00±0.00
PB	Avg. #	0.00	1,020.73±2,294.43	33,379.31±18,3229.66
	Avg. #Lits	0.00	24.95±27.60	105.21±109.99
	Avg. Coeff. Size	0.00	204.93±1,118.74	10.79±50.42

7.1 Benchmarks

To evaluate VeritasPBLib, we collected 1,803 pseudo-Boolean formulas from the PB Evaluation 2016.[4] These instances can be partitioned into formulas with (1) only clauses (279 instances), (2) clauses and cardinality constraints (772 instances) referred to as *Card* in what follows, (3) clauses and general PB constraints (444 instances) called *PB*, and (4) clauses, cardinality and general PB constraints (308 instances) called *Card+PB*. Since this work targets the verification of formulas with non-clausal constraints, we excluded the 279 pure CNF formula instances, as those can already be certified with existing techniques.

Table 1 shows some properties of the benchmarks used in the experimental results, namely, the average number of constraints, the average number of literals in each constraint, and the average size of coefficients associated with each literal. For each average value (*avg*), we also show the respective standard deviation (*std*) and denote it by $avg \pm std$. This information is shown for both cardinality constraints and PB constraints. Since the benchmark set is composed of instances from multiple domains, there is a large variation of values between instances. For example, the number of cardinality constraints for instances in the *Card* benchmark set ranges from 1 to 2,720, whereas the number of PB constraints for instances in the *PB* benchmark set ranges from 1 to 18,798. In the *Card+PB* benchmark set, we have an even larger dispersion with instances with 1 to 2,378,901 PB constraints and 1 to 75,582 cardinality constraints.

7.2 End-to-End Solving and Verification

Table 2 shows how VeritasPBLib can be used to generate a CNF formula that can be solved by Kissat and verified by VeriPB. For instances with cardinality constraints (*Card*), we use the sequential counter and totalizer encodings to translate those constraints to CNF. For instances with general PB constraints (*PB*), our translations use the adder network and generalized totalizer (*GTE*) encodings. Finally, for instances with both cardinality and general PB constraints (*Card+PB*), we use the sequential counter encoding for cardinality constraints and the adder network encoding for PB constraints, henceforth denoted by *Seq+Adder*. Even though other combinations of cardinality and PB encodings could be explored, the goal of this

[4]The benchmarks from the Pseudo-Boolean Evaluation 2016 are available at `http://www.cril.univ-artois.fr/PB16/`.

Table 2: *Number of translated, solved and verified instances for each encoding.*

			Translation		Solving			
					#Solved		#Verified	
Category	#Inst	Encoding	#CNF	#Veri	SAT	UNSAT	SAT	UNSAT
Card	772	Sequential	772	772	139	480	133	479
		Totalizer	772	772	139	475	130	474
PB	444	Adder	444	444	179	167	178	165
		GTE	425	414	164	162	150	151
Card+PB	308	Seq+Adder	306	296	134	152	128	151

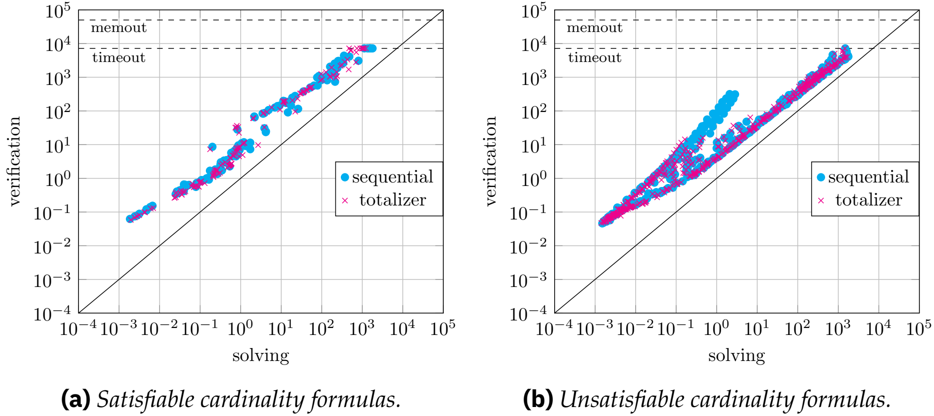
work is not to find the best-performing encoding but to show that we can verify the final result for a variety of encodings.

The column #*CNF* shows for how many instances VeritasPBLib successfully generated the CNF translation, which is almost all. The exceptions are 19 instances using *GTE* and 2 instances using the *Seq+Adder* encoding. In those cases, the number of clauses generated is too large and exceeds the resource limits used in our evaluation.

The column #*Veri* under *Translation* shows results for VeriPB verification of the translation certificate from VeritasPBLib. Except for a few instances for *GTE* and *Seq+Adder* yielding large proofs, VeriPB is successful. If the translation check passes, then this guarantees that the CNF encoding does not remove any solutions of the pseudo-Boolean formula.

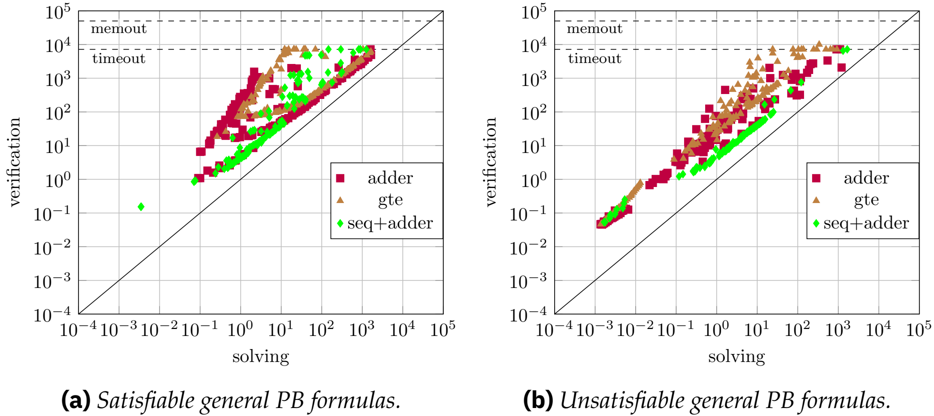
The columns #*Solved* and #*Verified* under *Solving* show how many instances can be solved by Kissat, and from those, how many can be verified by VeriPB. If a satisfiable formula is verified, then all clauses learned by Kissat are also valid for the original pseudo-Boolean formula, as is the satisfying assignment found. If an unsatisfiable formula is verified, then a correct proof of unsatisfiability for the PB formula has been produced.

We can verify 99% of the solved unsatisfiable instances, which shows that the current proof-of-concept approach is already practical in this setting. VeriPB proof logs can also be produced for satisfiable instances. These proof logs contain the derivations of all pseudo-Boolean constraints used in the solvers reasoning until a solution is found. Hence, it is possible to verify that the reasoning of the solver is sound, even if the instance is satisfiable. For satisfiable formulas we can verify that the reasoning was correct for 95% of the solved instances. However, even when VeriPB does not terminate within the time limit, we can still certify that the satisfying assignment found by the SAT solver is valid for the original PB formula. We note that there is still ample room for performance improvements in VeriPB—in particular, when it comes to verifying the *DRAT* proofs produced by the SAT solver, which do not even use pseudo-Boolean reasoning, but are simply clausal proofs syntactically rewritten in pseudo-Boolean format. Implementing backwards checking [GN03] and some minor engineering should get VeriPB close to the performance of DRAT-trim [WHH14] on *DRAT* proofs. Hence, there is no fundamental difficulty is improving the performance of VeriPB, but such work is



(a) *Satisfiable cardinality formulas.* **(b)** *Unsatisfiable cardinality formulas.*

Figure 6: *Comparing end-to-end solving and verification time for cardinality formulas.*



(a) *Satisfiable general PB formulas.* **(b)** *Unsatisfiable general PB formulas.*

Figure 7: *Comparing end-to-end solving and verification time for pseudo-Boolean formulas.*

mostly a matter of engineering and is fairly orthogonal to the contributions of this paper.

Figures 6 and 7 present the relationship between the end-to-end solving time (encoding the pseudo-Boolean formula to CNF and solving the resulting formula using KISSAT) and the time to check the resulting proof log for the end-to-end solving using VERIPB. It can be seen that even though we can verify most instances, verification is often considerably slower than solving. The time to verify an instance in proportion to solving it varies significantly. This is due to the verification of the proof generated by the SAT solver, since VERIPB has not been optimized to check such proofs.

Figure 6 compares end-to-end solving and verification time for formulas with only cardinality constraints and Figure 7 does the same comparison for formulas with general pseudo-Boolean constraints. We split the benchmarks between

satisfiable and unsatisfiable instances to analyze if the satisfiability of the formula affects the overhead of verification.

For the *sequential counter* encoding, verification of the proof for satisfiable instances takes on average 11.27 ± 6.98 times longer than solving and for unsatisfiable instances 18.30 ± 22.12 times longer. Even though verification times vary significantly for unsatisfiable instances, in the median satisfiable instances are checked within 8.45 times the solving time while unsatisfiable instances are checked within 5.16 times the solving time. The overhead for satisfiable instances may seem large, but note that VeriPB also checks if the derivations in the proof log of these instances are sound and not just the correctness of the result, which is the case in many occasions, e.g., the SAT competition.

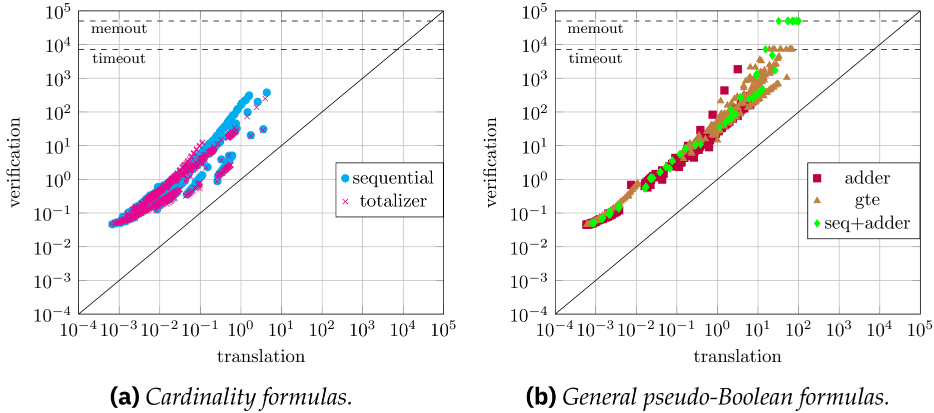
We observe a similar behavior with the *totalizer* encoding, where verifying the proof takes on average 11.30 ± 8.38 times longer than solving for satisfiable instances and 14.83 ± 14.54 times longer than solving for unsatisfiable instances. Similarly, there are quite different verification times for unsatisfiable instances, in the median there is an 8.62 times overhead for satisfiable instances while only having an overhead of 5.17 times for unsatisfiable instances.

For the general pseudo-Boolean formulas, we observe a higher verification overhead with respect to solving time. In particular, *GTE* has an average overhead of 54.67 ± 61.85 times for unsatisfiable instances and 89.88 ± 134.77 times for satisfiable instances, with a median overhead of 36.71 times for unsatisfiable instances and 18.68 times for satisfiable instances. A similar scenario applies with the *Adder* encoding with an average 29.69 ± 28.41 times overhead for unsatisfiable instances and 54.00 ± 99.29 times for satisfiable instances, with a median overhead of 28.42 times for unsatisfiable instances and 5.44 times for satisfiable instances. Verifying the results reported by the SAT solver are harder for formulas containing PB constraints than for formulas containing cardinality constraints.

When considering formulas with both cardinality and pseudo-Boolean constraints, the observed overhead is smaller than for the other formulas with an average overhead of 7.89 ± 9.44 times for unsatisfiable instances and 13.11 ± 19.39 times for satisfiable instances, with a median overhead of 4.33 times for unsatisfiable instances and 5.21 times for satisfiable instances.

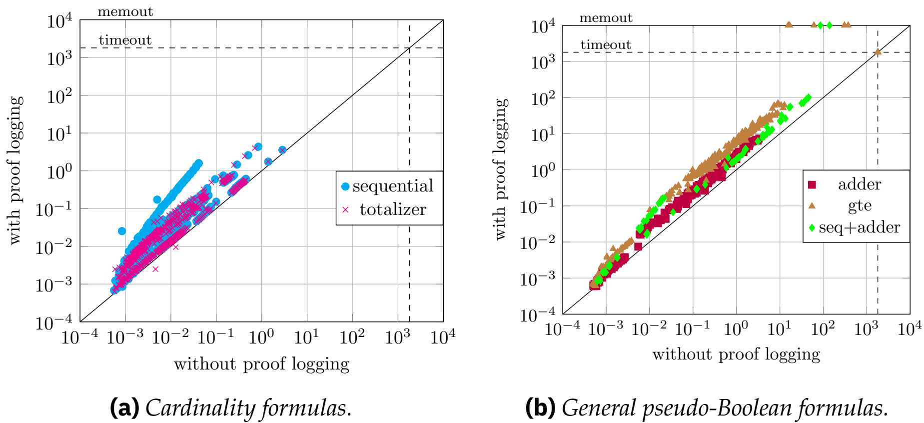
7.3 Translation and Verification

For a more detailed discussion of our results, let us first turn to the certified translation. Figure 8 compares the time for VeritasPBLib to generate the CNF translation and VeriPB to verify it. The verification overhead is far from negligible, but is not unreasonable. Over all encodings, for 75% of benchmarks verification takes at most 49 times longer than translation, and for 98% of benchmarks at most 100 times longer. While some overhead is natural, since the translation algorithm can just output a claimed proof while the verifier needs to perform the calculations to actually check it, our experiments do show that there is room for further improvements in efficiency both for the verifier and for the proof logging methods.



(a) *Cardinality formulas.* **(b)** *General pseudo-Boolean formulas.*

Figure 8: *Comparison between CNF translation and verification of proof logging.*



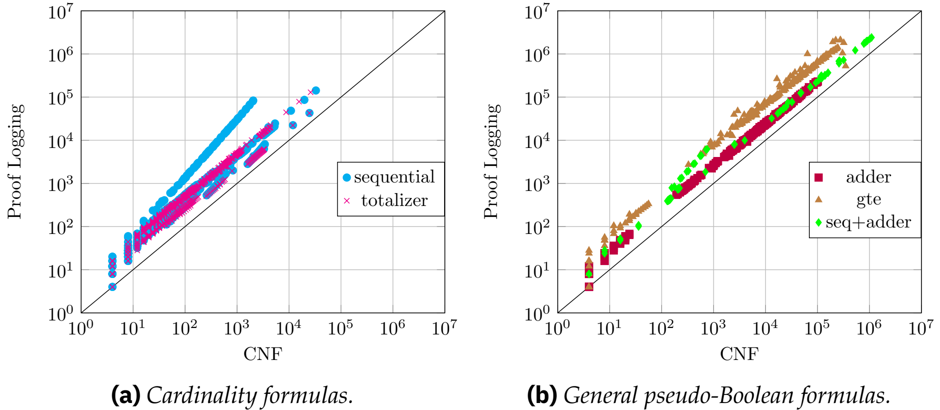
(a) *Cardinality formulas.* **(b)** *General pseudo-Boolean formulas.*

Figure 9: *Comparison of running times for CNF translation with and without proof logging.*

7.4 Overhead of Proof Logging

Figure 9 shows the overhead of proof logging when translating the pseudo-Boolean formulas to CNF. For the majority of the instances, the overhead is not too significant, and formulas with just cardinality constraints can still be translated under 10 seconds, while formulas with PB constraints can be translated under 100 seconds. The average overhead in running time for proof logging is a factor of 2–3 for all encodings except *GTE*, which incurs around a factor 5 in overhead. However, since translation is fast for the majority of instances, the additional overhead of proof logging is not an issue when translating the pseudo-Boolean formulas to CNF.

The proof logging overhead can be explained by the proofs being larger than the generated CNF formulas, as shown in Figure 10. For most instances the proof size seems to be within a constant factor of the CNF formula size, but for



(a) *Cardinality formulas.*

(b) *General pseudo-Boolean formulas.*

Figure 10: *Comparison between CNF file size and proof logging file size in KiB.*

a collection of crafted vertex cover problems [EGNV18] the sequential counter encoding turns out to require proofs of super-linear size. These instances contain a cardinality constraint enforcing a constant fraction of the variables in the formula to be false, which is a worst-case scenario for the sequential counter encoding. While the number of clauses in the CNF translation and the number of proof steps are quadratic in the number of literals in the constraint, each reification step in the unary sum derivation in Proposition 2 introduces a constraint of linear size, making the total proof size cubic while the size of the CNF encoding remains quadratic. It would be desirable to find a more efficient derivation that only requires quadratic proof size in the number of literals in the constraint.

Additionally, there were 6 instances where VeritasPBLib had memory outs, as the whole proof for the translation is stored in memory. This could be improved in the future by only storing the proof for one constraint at a time in VeritasPBLib.

7.5 Comparison with PB Solvers

In Table 3, we report results on how VeritasPBLib together with Kissat used as a pseudo-Boolean solver using the sequential counter and adder encodings compares to state-of-the-art PB solvers, namely, MiniSat+ [ES06], Gurobi [Opt22] (version 9.5.1), NaPS [SN15] (version 1.02b), Open-WBO [MML14], RoundingSat [EN18] (commit b5de84d), and Sat4j [LP10] (version v20220212). All solvers were run with their respective default configurations.

The approach of VeritasPBLib+Kissat to transform a PB formula into CNF and using a CDCL SAT solver to solve the resulting formula is also made by other PB solvers such as MiniSat+, NaPS, and Open-WBO. However, the encodings used in these solvers differ, and VeritasPBLib uses a more recent SAT solver (Kissat). For this benchmark set, VeritasPBLib+Kissat outperforms other PB solvers in the *Card+PB* and *PB* categories while being third in the *Card* category. Note that for Sat4j, we ran the default version, which has native support for PB constraints and does not translate them to CNF but still uses a SAT solver to solve the resulting

Table 3: *Number of solved instances by each PB solver*

Solver	Card	Card+PB	PB	Total
MiniSat+	490	269	323	1,082
Gurobi	610	256	230	1,096
NaPS	555	265	283	1,103
Open-WBO	600	275	316	1,191
RoundingSat	663	270	273	1,206
Sat4j	455	265	275	995
VeritasPBLib+Kissat	619	286	346	1,251

formula.[5] However, since Sat4j is written in Java and the underlying SAT solver is not as powerful as the other solvers, its performance is worse when compared to the other solvers.

Instead of using resolution like the SAT-based approaches, RoundingSat uses stronger pseudo-Boolean reasoning in the form of cutting planes. For this benchmark set, RoundingSat performed better than SAT-based solvers for the *Card* category but worse for the *Card+PB* and *PB* categories.

Figure 11 shows a cumulative plot with the runtime comparison of pseudo-Boolean solvers. We can observe that a majority of the instances are solved after a few seconds. Overall, VeritasPBLib+Kissat not only provides certificates that can be checked by VeriPB, but is also one of the best approaches to solving pseudo-Boolean decision problems.

7.6 Certifying MaxSAT Optimal Values

Maximum Satisfiability (MaxSAT) [BJM21] is the optimization counterpart of SAT, where the goal is to maximize the number of satisfied clauses. The MaxSAT problem can be generalized to have *hard* and *soft* clauses, where hard clauses *must* be satisfied and soft clauses may or may not be satisfied. Each soft clause has a weight associated with it that corresponds to the cost of falsifying that soft clause. For the general MaxSAT problem, the optimization goal becomes to maximize the sum of the weights of the satisfied soft clauses. This optimization problem can also be viewed as minimizing the sum of the weights of falsified soft clauses. An optimal value of a MaxSAT formula corresponds to the minimal sum of the weights of the falsified soft clauses.

The annual MaxSAT Evaluation[6] focus on evaluating the current state-of-the-art in MaxSAT solvers. It has two main categories: (1) unweighted, where all soft clauses have a weight of 1, and (2) weighted, where soft clauses have a weight between 1 and 2^{63}. In contrast to the SAT competition, the results of MaxSAT solvers are not verified since there is no verification tool for MaxSAT. Instead, the optimal solution claimed by the solvers is checked to be a valid solution (i.e.,

[5] Sat4j best-performing version for PB formulas is to run a cutting-planes-based solver with a resolution solver in parallel. We did not present results for this version since we only run single-threaded solvers, and this is not the default configuration of Sat4j. However, even with this version Sat4j consistently performs worse than other cutting-planes-based solvers like RoundingSat [EN18].

[6] `https://maxsat-evaluations.github.io/`

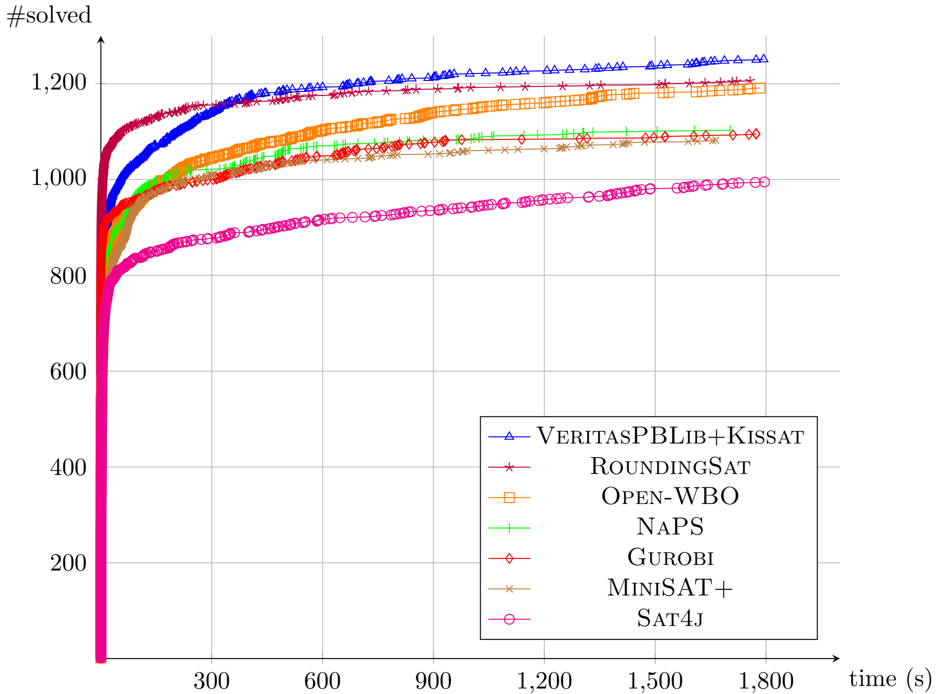


Figure 11: *Cumulative plot with runtime comparison of PB solvers.*

satisfies all hard clauses, and the optimal value corresponds to the sum of the weights of the falsified soft clauses), and any of the competing solvers found no solution with a smaller value. However, this procedure does not give any correctness guarantees. It has occurred in previous years that a single solver found a (claimed) optimal solution for an instance, but this solution was later found not to be optimal.[7]

Even though VeritasPBLib+Kissat cannot be used to show the correctness of the solving procedure of a MaxSAT solver, it may be used to certify that the optimal value of a given instance is correct. Given a MaxSAT formula F and its respective optimal value k, we turn the task of proving optimality of a MaxSAT instance F into solving a PB decision instance F_{PB} that encodes that no smaller optimal value exists for F. Let $F = F_h \cup F_s$ be a MaxSAT formula, where F_h represents h hard clauses and F_s represents s soft clauses. Let the weight associated with each soft clause $D_j \in F_s$ be a_j and k the optimal value of F. We construct F_{PB} as follows.

- Each clause $C = (\ell_1 \vee \ldots \vee \ell_n) \in F_h$ is added in pseudo-Boolean form to F_{PB}: $\ell_1 + \ldots + \ell_n \geq 1$;

- For each clause $D_j = (\ell_1 \vee \ldots \vee \ell_m) \in F_s$, we introduce a fresh variable b_j and add the clause in pseudo-Boolean form to F_{PB}: $\ell_1 + \ldots + \ell_m + b_j \geq 1$;

[7] `http://www.maxsat.udl.cat/15/results/index.html`

Table 4: *Number of translated, solved and verified instances for each encoding when certifying the results from the MaxSAT Evaluation 2022.*

			Translation		Solving	
Category	#Inst	Encoding	#CNF	#Verified	#Solved	#Verified
Unweighted	468	Sequential	411	333	329	265
		Totalizer	448	408	358	307
Weighted	473	Adder	455	346	193	139
		GTE	262	186	221	169

- Add a PB constraint to F_{PB} that restricts the sum of the weights of falsifying soft clauses to be at most $k-1$: $a_1b_1 + \ldots + a_sb_s \leq k-1$.

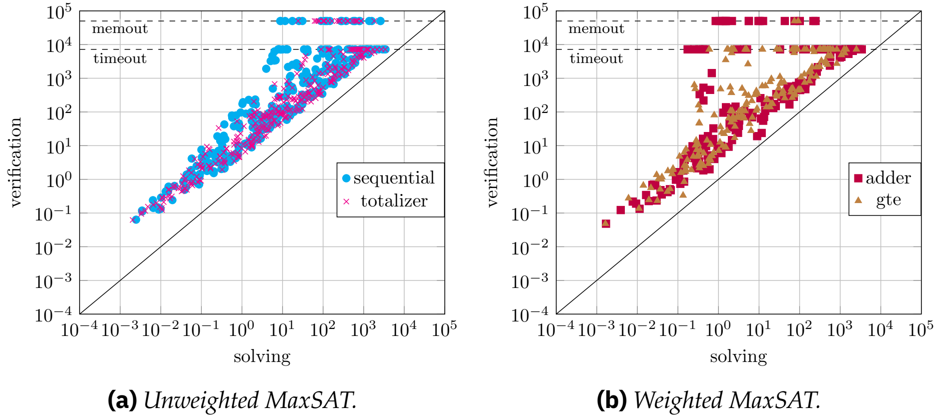
We can use VeritasPBLib to translate F_{PB} to CNF, use Kissat to solve the resulting formula, and then verify the results with VeriPB. If the formula is unsatisfiable, we can certify that there is no solution with an objective value smaller than k. The solution that results in the optimal value k has already been tested to be a valid solution. Therefore, if we prove that no better solution exists, we can show that the optimal value returned by the MaxSAT solver is correct.

To evaluate how effective VeritasPBLib+Kissat is to certify the results of the MaxSAT Evaluation 2022,[8] we used the instances for which at least one solver found an optimal solution, namely, 468 unweighted instances and 473 weighted instances. Similarly to the previous experiments, we used a memory limit of 15 GB, a time limit of 7,200 seconds for verifying the proof with VeriPB, and a time limit of 1,800 seconds for CNF translation with VeritasPBLib. We increase the SAT solving time for Kissat to 3,600 seconds to match the time limit used in the MaxSAT Evaluation.

Table 4 shows the number of instances for which VeritasPBLib+Kissat could verify the optimal value. The information is split into *Translation* and *Solving*. The column #*CNF* presents the number of instances where VeritasPBLib successfully generated a CNF formula from the pseudo-Boolean formula F_{PB}. Note that all F_{PB} are unsatisfiable, and each of them only contains either a *single cardinality constraint* (in the case of unweighted) or a *single PB constraint* (in the case of weighted), with the remaining constraints being clauses. The column *Verified* under *Translation* shows how many instances were verified by VeriPB for the proof logging certificate generated by VeritasPBLib when translating each F_{PB} to CNF. The columns #*Solved* and #*Verified* under *Solving* present how many instances were solved by Kissat, and from those, how many were verified by VeriPB.

For the unweighted category, we can observe that 20 instances cannot be translated to CNF with the totalizer encoding, which shows the blowup in the CNF encoding when translating a single cardinality constraint to CNF. Overall, Kissat can solve 358 out of 468 instances (76%) using this approach. Even though this is a large percentage of instances, it does not match the performance of the best MaxSAT solvers, since they are able to prove optimality by solving a different CNF formula that is equivalent but simpler than the one we are solving with our

[8] `https://maxsat-evaluations.github.io/2022/`



(a) *Unweighted MaxSAT.* **(b)** *Weighted MaxSAT.*

Figure 12: *Comparison between end-to-end solving and verification time for the MaxSAT Evaluation 2022 instances that can be solved by* VeritasPBLib+Kissat*.*

approach. Nevertheless, by using VeritasPBLib+Kissat, we can certify 307 out of 358 instances (86%), which shows the positive result that if we can solve the formula with Kissat, we are likely able to certify the results. We can observe a similar behavior with the sequential counter encoding, albeit with worse performance, since this encoding is not as efficient for solving instances with a single large cardinality constraint.

For the weighted category, when using the GTE encoding, we can translate 262 out of 473 instances (55%) to CNF and when using the adder network encoding, we can translate 455 out of 473 instances (96%). This large difference is due to the exponential growth of the GTE encoding concerning the size of the weights, while the adder network encoding only grows linearly. However, despite this significant difference, Kissat can solve more instances with the GTE encoding than with the adder network encoding since the GTE encoding is arc consistent [Gen02] and the adder network encoding is not. When the formula can be translated to CNF using the GTE encoding, then Kissat can solve it in 221 out of 262 cases (84%). Overall, using VeritasPBLib+Kissat, we can certify 169 out of 221 instances (76%) when using the GTE encoding and 139 out of 193 instances (72%) when using the adder network encoding. If we consider instances that can be solved or certified by either using the adder network encoding or the GTE encoding, then Kissat can solve 247 instances, and VeriPB can certify 199 of those. This shows that the adder network encoding and GTE encoding are complementary and using both encoding can increase the number of certified instances.

Figure 12 compares the end-to-end time between translation plus solving with VeritasPBLib+Kissat and verifying the proof using VeriPB. As in Section 7.2, we can observe that although we can verify most instances that the SAT solver solves, verification is often considerably slower than solving the problem.

8 Concluding Remarks

In this work, we develop a general framework for certified translations of linear pseudo-Boolean constraints into CNF using cutting-planes-based proof logging. Since our method is a strict extension of the *DRAT* proof logging method used by conflict-driven clause learning (CDCL) SAT solvers, the proof for the PB-to-CNF translation can be combined with a SAT solver *DRAT* proof log to provide, for the first time, end-to-end verification for SAT-based pseudo-Boolean solvers. Our use of the cutting planes method is not only crucial to deal with the pseudo-Boolean format of the input, but the expressivity of the 0-1 linear constraints also allows us to certify the correctness of the translation to CNF in a concise and elegant way.

While there is still room for performance improvements in proof logging and verification, the experimental evaluation shows that our approach is feasible in practice. We believe that the generality of our method, which expresses the proof logging steps in terms of simple operations on a graph representation of the PB-to-CNF translation, is an important aspect of our work. To demonstrate this generality of our framework we show how to do proof logging for the sequential counter, binary adder and (generalized) totalizer encodings. We are optimistic that our framework can also be used for the watchdog encoding [BBR09], which builds on top of the totalizer encoding. It is less clear whether the graph representation can also be used in an elegant way to capture some of the *sorting networks* encodings found to be particularly efficient in [ES06], such as the *odd-even merge sorters* [Bat68] used in MiniSat+, or BDD-based encodings [Bry86, ES06], or whether more ad-hoc pseudo-Boolean proof logging methods would be needed for such encodings.

As discussed already in the introduction, our paper does not quite reach the goal of certifying *equivalence* of the original pseudo-Boolean formula F and the CNF translation F'. In one direction, it is clear that as long as F' is derived from F using cutting planes with reification, any satisfying assignment α to F yields a unique extended assignment $\alpha' \supseteq \alpha$ satisfying F' by giving all newly introduced variables the values determined by the reification rules (5a)–(5b). In the other direction, however, we do not formally establish that the CNF translation F' is as strong as the original pseudo-Boolean formula F in the sense that any satisfying assignment α' for F' is guaranteed to also satisfy F. As a quick technical detour, one way of achieving such guarantees would be, after having derived all clauses in F', to erase all constraints in F using the "checked deletion" rule in [BGMN22]. This is certainly doable in principle, but we currently know of no clean and simple way to formalize this in our graph-based translation framework. This is therefore another problem that we have to leave as future research.

Our work on proof logging for PB-to-CNF translations has also uncovered some technical questions that, to the best of our knowledge, have not been studied in the literature before, but would seem to merit further investigations. A common theme is that these questions revolve around possible trade-offs between encoding strength and encoding size, as explained below.

In our proofs of correctness for the order encoding, the derivation of binary clauses enforcing $z_i \geq z_{i+1}$ play a key role in the derivations, but are not included in the final CNF translation. This is a little bit surprising, since it would seem that such clauses would improve propagation and hence potentially help the SAT

solver discover more facts. On the other hand, it is not clear how the presence of such clauses in the solver trail would affect the conflict analysis. Thus it would be interesting to study whether including such clauses in the PB-to-CNF translation would affect the SAT solving process in any systematic way.

Another question concerns the translation of cardinality constraints. When given a constraint $\sum_{i=1}^{n} a_i \ell_i \geq k$ such that $\sum_{i=1}^{n} a_i - k < k$, the PB-to-CNF translation instead uses the equivalent constraint $\sum_{i=1}^{n} a_i \overline{\ell_i} \leq \sum_{i=1}^{n} a_i - k$ because it introduces fewer auxiliary variables. On the other hand, it seems that the presence of auxiliary variables encoding partial information about constraints is precisely what allows SAT-based pseudo-Boolean solvers to compete with, and not seldom outperform, cutting-planes-based solvers [EGNV18]. And perhaps there could be a reason why the problem at hand was encoded with a greater-than-or-equal constraint rather than less-than-or-equal. It would seem relevant to investigate if there is a trade-off here between propagation strength (potentially leading to more efficient SAT solver search) and encoding size (potentially slowing down the solver due to the increased number of auxiliary variables).

A final question regarding CNF translation of circuits is whether it is better to encode propagations in both directions or only in one direction. If a circuit encodes the evaluation of a PB constraint $\sum_i a_i \ell_i \geq A$, then one direction of propagation is that any assignment to the literals ℓ_i making the constraint true should make the output gate of the circuit evaluate to true. The other direction of propagation is that any literal assignment violating the constraint should make the output gate of the circuit evaluate to false. In our proofs of correctness we generate constraints encoding both types of propagation, but it seems that in the final CNF translation it is most common to include only clauses enforcing one of the directions. This cuts the encoding size in half, but at the price of losing propagation power of the encoding. It would be quite interesting to investigate how enforcing two-way propagation or only one-way propagation affects the efficiency of the SAT solver search.

Concluding this section, we wish to emphasize that we view certified translations to CNF of pseudo-Boolean decision problems as only a first step. In the conference version of this paper, we expressed optimism that the techniques developed in this work should also be possible to extend to *core-guided MaxSAT solving* [FM06, MHL+13], including proof logging support for derivation of clauses added during core extraction and objective function reformulation, and such results have very recently been announced in [BBN+23]. While designing efficient proof logging for other MaxSAT approaches such as *implicit hitting sets (IHS)* [DB11] and *abstract cores* [BBP20] seems more challenging, we are hopeful that our work could lead to a unified proof logging method for all modern MaxSAT solving techniques, and also for pseudo-Boolean optimization using cutting-planes-based reasoning as in [DGN21, DGD+21, EN18, LP10, SBJ21, SBJ22].

Acknowledgements

The authors wish to acknowledge helpful and stimulating discussions with Bart Bogaerts and Ciaran McCreesh. We are particularly grateful to Bart for

sharing the manuscript [VWB22] using a very elegant reification technique that we wish we would have thought of, and we believe it would be worth exploring whether similar ideas could be used in our framework to improve the efficiency of verification. We are also thankful for discussions and feedback at the workshops *Pragmatics of SAT* and *Proof eXchange for Theorem Proving* in 2021 and the conference *SAT* in 2022, where a preliminary version of this work was presented. We would like to extend a special thanks to the *SAT '22* and *JAIR* anonymous reviewers for a wealth of comments and questions that helped to improve this manuscript considerably.

This work has been financially supported by the Swedish Research Council grant 2016-00782, the Independent Research Fund Denmark grant 9040-00389B, the National Science Foundation award CCF-1762363, the Amazon Research Award, and the Wallenberg AI, Autonomous Systems and Software Program (WASP) funded by the Knut and Alice Wallenberg Foundation.

References

[AGJ+18] Özgür Akgün, Ian P. Gent, Christopher Jefferson, Ian Miguel, and Peter Nightingale. Metamorphic testing of constraint solvers. In *Proceedings of the 24th International Conference on Principles and Practice of Constraint Programming (CP '18)*, volume 11008 of *Lecture Notes in Computer Science*, pages 727–736. Springer, August 2018.

[Bar95] Peter Barth. A Davis-Putnam based enumeration algorithm for linear pseudo-Boolean optimization. Technical Report MPI-I-95-2-003, Max-Planck-Institut für Informatik, January 1995.

[Bat68] Kenneth E. Batcher. Sorting networks and their applications. In *Proceedings of the Spring Joint Computer Conference of the American Federation of Information Processing Societies (AFIPS '68)*, volume 32, pages 307–314, April 1968.

[BB03] Olivier Bailleux and Yacine Boufkhad. Efficient CNF encoding of Boolean cardinality constraints. In *Proceedings of the 9th International Conference on Principles and Practice of Constraint Programming (CP '03)*, volume 2833 of *Lecture Notes in Computer Science*, pages 108–122. Springer, September 2003.

[BB21] Lee A. Barnett and Armin Biere. Non-clausal redundancy properties. In *Proceedings of the 28th International Conference on Automated Deduction (CADE-28)*, volume 12699 of *Lecture Notes in Computer Science*, pages 252–272. Springer, July 2021.

[BBH22] Randal E. Bryant, Armin Biere, and Marijn J. H. Heule. Clausal proofs for pseudo-Boolean reasoning. In *Proceedings of the 28th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS '22)*, volume 13243 of *Lecture Notes in Computer Science*, pages 443–461. Springer, April 2022.

[BBHJ13] Adrian Balint, Anton Belov, Marijn JH Heule, and Matti Järvisalo. Proceedings of sat competition 2013: Solver and benchmark descriptions, 2013.

[BBN+23] Jeremias Berg, Bart Bogaerts, Jakob Nordström, Andy Oertel, and Dieter Vandesande. Certified core-guided MaxSAT solving. Submitted manuscript, March 2023.

[BBP20] Jeremias Berg, Fahiem Bacchus, and Alex Poole. Abstract cores in implicit hitting set MaxSat solving. In *Proceedings of the 23rd International Conference on Theory and Applications of Satisfiability Testing (SAT '20)*, volume 12178 of *Lecture Notes in Computer Science*, pages 277–294. Springer, July 2020.

[BBR09] Olivier Bailleux, Yacine Boufkhad, and Olivier Roussel. New encodings of pseudo-Boolean constraints into CNF. In *Proceedings of the 12th International Conference on Theory and Applications of Satisfiability Testing (SAT '09)*, volume 5584 of *Lecture Notes in Computer Science*, pages 181–194. Springer, June 2009.

[BCH21] Seulkee Baek, Mario Carneiro, and Marijn J. H. Heule. A flexible proof format for SAT solver-elaborator communication. In *Proceedings of the 27th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS '21)*, volume 12651 of *Lecture Notes in Computer Science*, pages 59–75. Springer, March-April 2021.

[BFFH20] Armin Biere, Katalin Fazekas, Mathias Fleury, and Maximillian Heisinger. CaDiCaL, Kissat, Paracooba, Plingeling and Treengeling entering the SAT Competition 2020. In Tomas Balyo, Nils Froleyks, Marijn Heule, Markus Iser, Matti Järvisalo, and Martin Suda, editors, *Proc. of SAT Competition 2020 – Solver and Benchmark Descriptions*, volume B-2020-1 of *Department of Computer Science Report Series B*, pages 51–53. University of Helsinki, 2020.

[BGMN22] Bart Bogaerts, Stephan Gocht, Ciaran McCreesh, and Jakob Nordström. Certified symmetry and dominance breaking for combinatorial optimisation. In *Proceedings of the 36th AAAI Conference on Artificial Intelligence (AAAI '22)*, pages 3698–3707, February 2022.

[BHvMW21] Armin Biere, Marijn J. H. Heule, Hans van Maaren, and Toby Walsh, editors. *Handbook of Satisfiability*, volume 336 of *Frontiers in Artificial Intelligence and Applications*. IOS Press, 2nd edition, February 2021.

[Bie06] Armin Biere. Tracecheck. `http://fmv.jku.at/tracecheck/`, 2006.

[BJM21] Fahiem Bacchus, Matti Järvisalo, and Ruben Martins. Maximum satisfiabiliy. In Armin Biere, Marijn Heule, Hans van Maaren, and Toby Walsh, editors, *Handbook of Satisfiability - Second Edition*, volume 336 of *Frontiers in Artificial Intelligence and Applications*, pages 929–991. IOS Press, 2021.

[BLB10] Robert Brummayer, Florian Lonsing, and Armin Biere. Automated testing and debugging of SAT and QBF solvers. In *Proceedings of the 13th International Conference on Theory and Applications of Satisfiability Testing (SAT '10)*, volume 6175 of *Lecture Notes in Computer Science*, pages 44–57. Springer, July 2010.

[BN21] Samuel R. Buss and Jakob Nordström. Proof complexity and SAT solving. In Biere et al. [BHvMW21], chapter 7, pages 233–350.

[Bry86] Randal E. Bryant. Graph-based algorithms for Boolean function manipulation. *IEEE Transactions on Computers*, C-35(8):677–691, August 1986.

[CCT87] William Cook, Collette Rene Coullard, and György Turán. On the complexity of cutting-plane proofs. *Discrete Applied Mathematics*, 18(1):25–38, November 1987.

[CFHH+17] Luís Cruz-Filipe, Marijn J. H. Heule, Warren A. Hunt, Matt Kaufmann, and Peter Schneider-Kamp. Efficient certified RAT verification. In *Proceedings of the 26th International Conference on Automated Deduction (CADE-26)*, volume 10395 of *LNCS*, pages 220–236. Springer, 2017.

[CFMSSK17] Luís Cruz-Filipe, Joao Marques-Silva, and Peter Schneider-Kamp. Efficient certified resolution proof checking. In *Proceedings of the 23rd International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS '17)*, volume 10205 of *LNCS*, pages 118–135. Springer, 2017.

[CGS17] Kevin K. H. Cheung, Ambros M. Gleixner, and Daniel E. Steffy. Verifying integer programming results. In *Proceedings of the 19th International Conference on Integer Programming and Combinatorial Optimization (IPCO '17)*, volume 10328 of *Lecture Notes in Computer Science*, pages 148–160. Springer, June 2017.

[CKSW13] William Cook, Thorsten Koch, Daniel E. Steffy, and Kati Wolter. A hybrid branch-and-bound approach for exact rational mixed-integer programming. *Mathematical Programming Computation*, 5(3):305–344, September 2013.

[DB11] Jessica Davies and Fahiem Bacchus. Solving MAXSAT by solving a sequence of simpler SAT instances. In *Proceedings of the 17th International Conference on Principles and Practice of Constraint Programming (CP '11)*, volume 6876 of *Lecture Notes in Computer Science*, pages 225–239. Springer, September 2011.

[DGD+21] Jo Devriendt, Stephan Gocht, Emir Demirović, Jakob Nordström, and Peter Stuckey. Cutting to the core of pseudo-Boolean optimization: Combining core-guided search with cutting planes reasoning. In *Proceedings of the 35th AAAI Conference on Artificial Intelligence (AAAI '21)*, pages 3750–3758, February 2021.

[DGN21] Jo Devriendt, Ambros Gleixner, and Jakob Nordström. Learn to relax: Integrating 0-1 integer linear programming with pseudo-Boolean conflict-driven search. *Constraints*, 26(1–4):26–55, October 2021. Preliminary version in *CPAIOR '20*.

[EG21] Leon Eifler and Ambros Gleixner. A computational status update for exact rational mixed integer programming. In *Proceedings of the 22nd International Conference on Integer Programming and Combinatorial Optimization (IPCO '21)*, volume 12707 of *Lecture Notes in Computer Science*, pages 163–177. Springer, May 2021.

[EGMN20] Jan Elffers, Stephan Gocht, Ciaran McCreesh, and Jakob Nordström. Justifying all differences using pseudo-Boolean reasoning. In *Proceedings of the 34th AAAI Conference on Artificial Intelligence (AAAI '20)*, pages 1486–1494, February 2020.

[EGNV18] Jan Elffers, Jesús Giráldez-Cru, Jakob Nordström, and Marc Vinyals. Using combinatorial benchmarks to probe the reasoning power of pseudo-Boolean solvers. In *Proceedings of the 21st International Conference on Theory and Applications of Satisfiability Testing (SAT '18)*, volume 10929 of *Lecture Notes in Computer Science*, pages 75–93. Springer, July 2018.

[EN18] Jan Elffers and Jakob Nordström. Divide and conquer: Towards faster pseudo-Boolean solving. In *Proceedings of the 27th International Joint Conference on Artificial Intelligence (IJCAI '18)*, pages 1291–1299, July 2018.

[ES06] Niklas Eén and Niklas Sörensson. Translating pseudo-Boolean constraints into SAT. *Journal on Satisfiability, Boolean Modeling and Computation*, 2(1-4):1–26, March 2006.

[FM06] Zhaohui Fu and Sharad Malik. On solving the partial MAX-SAT problem. In *Proceedings of the 9th International Conference on Theory and Applications of Satisfiability Testing (SAT '06)*, volume 4121 of *Lecture Notes in Computer Science*, pages 252–265. Springer, August 2006.

[Gen02] Ian P. Gent. Arc consistency in SAT. In Frank van Harmelen, editor, *Proceedings of the 15th European Conference on Artificial Intelligence, ECAI'2002, Lyon, France, July 2002*, pages 121–125. IOS Press, 2002.

[GMM+20] Stephan Gocht, Ross McBride, Ciaran McCreesh, Jakob Nordström, Patrick Prosser, and James Trimble. Certifying solvers for clique and maximum common (connected) subgraph problems. In *Proceedings of the 26th International Conference on Principles and Practice of Constraint Programming (CP '20)*, volume 12333 of *Lecture Notes in Computer Science*, pages 338–357. Springer, September 2020.

[GMN20] Stephan Gocht, Ciaran McCreesh, and Jakob Nordström. Subgraph isomorphism meets cutting planes: Solving with certified solutions. In *Proceedings of the 29th International Joint Conference on Artificial Intelligence (IJCAI '20)*, pages 1134–1140, July 2020.

[GMN22] Stephan Gocht, Ciaran McCreesh, and Jakob Nordström. An auditable constraint programming solver. In *Proceedings of the 28th International Conference on Principles and Practice of Constraint Programming (CP '22)*, volume 235 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 25:1–25:18, August 2022.

[GMNO22] Stephan Gocht, Ruben Martins, Jakob Nordström, and Andy Oertel. Experimental repository for "Certified CNF translations for pseudo-Boolean solving". Available at `https://doi.org/10.5281/zenodo.6610581`, June 2022.

[GN03] Evgueni Goldberg and Yakov Novikov. Verification of proofs of unsatisfiability for CNF formulas. In *Proceedings of the Conference on Design, Automation and Test in Europe (DATE '03)*, pages 886–891, March 2003.

[GN21] Stephan Gocht and Jakob Nordström. Certifying parity reasoning efficiently using pseudo-Boolean proofs. In *Proceedings of the 35th AAAI Conference on Artificial Intelligence (AAAI '21)*, pages 3768–3777, February 2021.

[GSD19] Xavier Gillard, Pierre Schaus, and Yves Deville. SolverCheck: Declarative testing of constraints. In *Proceedings of the 25th International Conference on Principles and Practice of Constraint Programming (CP '19)*, volume 11802 of *Lecture Notes in Computer Science*, pages 565–582. Springer, October 2019.

[HHW13a] Marijn J. H. Heule, Warren A. Hunt Jr., and Nathan Wetzler. Trimming while checking clausal proofs. In *Proceedings of the 13th International Conference on Formal Methods in Computer-Aided Design (FMCAD '13)*, pages 181–188, October 2013.

[HHW13b] Marijn J. H. Heule, Warren A. Hunt Jr., and Nathan Wetzler. Verifying refutations with extended resolution. In *Proceedings of the 24th International Conference on Automated Deduction (CADE-24)*, volume 7898 of *Lecture Notes in Computer Science*, pages 345–359. Springer, June 2013.

[HMS12] Steffen Hölldobler, Norbert Manthey, and Peter Steinke. A compact encoding of pseudo-Boolean constraints into SAT. In *Proceedings of KI 2012: Advances in Artificial Intelligence, the 35th Annual German Conference on AI*, volume 7526 of *Lecture Notes in Computer Science*, pages 107–118. Springer, 2012.

[JMM15] Saurabh Joshi, Ruben Martins, and Vasco M. Manquinho. Generalized totalizer encoding for pseudo-Boolean constraints. In *Proceedings of the 21st International Conference on Principles and Practice of Constraint Programming (CP '15)*, volume 9255 of *Lecture Notes in Computer Science*, pages 200–209. Springer, August-September 2015.

[KB21] Daniela Kaufmann and Armin Biere. AMulet 2.0 for verifying multiplier circuits. In *Proceedings of the 27th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS '21)*, volume 12652 of *Lecture Notes in Computer Science*, pages 357–364. Springer, March-April 2021.

[KBBN22] Daniela Kaufmann, Paul Beame, Armin Biere, and Jakob Nordström. Adding dual variables to algebraic reasoning for circuit verification. In *Proceedings of the 25th Design, Automation and Test in Europe Conference (DATE '22)*, pages 1435–1440, March 2022.

[KFB20] Daniela Kaufmann, Mathias Fleury, and Armin Biere. The proof checkers Pacheck and Pastèque for the practical algebraic calculus. In *Proceedings of the 20th Conference on Formal Methods in Computer-Aided Design (FMCAD '20)*, pages 264–269, September 2020.

[LP10] Daniel Le Berre and Anne Parrain. The Sat4j library, release 2.2. *Journal on Satisfiability, Boolean Modeling and Computation*, 7:59–64, July 2010.

[MHL+13] António Morgado, Federico Heras, Mark H. Liffiton, Jordi Planes, and João P. Marques-Silva. Iterative and core-guided MaxSAT solving: A survey and assessment. *Constraints*, 18(4):478–534, October 2013.

[MML14] Ruben Martins, Vasco M. Manquinho, and Inês Lynce. Open-WBO: A modular MaxSAT solver. In *Proceedings of the 17th International Conference on Theory and Applications of Satisfiability Testing (SAT '14)*, volume 8561 of *Lecture Notes in Computer Science*, pages 438–445. Springer, July 2014.

[MMNS11] Ross M. McConnell, Kurt Mehlhorn, Stefan Näher, and Pascal Schweitzer. Certifying algorithms. *Computer Science Review*, 5(2):119–161, May 2011.

[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 Design Automation Conference (DAC '01)*, pages 530–535, June 2001.

[MS99] João P. Marques-Silva and Karem A. Sakallah. GRASP: A search algorithm for propositional satisfiability. *IEEE Transactions on Computers*, 48(5):506–521, May 1999. Preliminary version in *ICCAD '96*.

[Opt22] Gurobi Optimization. Gurobi Optimization. Available at `https://www.gurobi.com/`, 2022.

[PS15] Tobias Philipp and Peter Steinke. PBLib – a library for encoding pseudo-Boolean constraints into CNF. In *Proceedings of the 18th International Conference on Theory and Applications of Satisfiability Testing (SAT '15)*, volume 9340 of *Lecture Notes in Computer Science*, pages 9–16. Springer, September 2015.

[RBK+18] Daniela Ritirc, Armin Biere, Manuel Kauers, A Bigatti, and M Brain. A practical polynomial calculus for arithmetic circuit verification. In *3rd International Workshop on Satisfiability Checking and Symbolic Computation (SC2 '18)*, pages 61–76, 2018.

[RM16] Olivier Roussel and Vasco M. Manquinho. Input/output format and solver requirements for the competitions of pseudo-Boolean solvers. Revision 2324. Available at `http://www.cril.univ-artois.fr/PB16/format.pdf`, January 2016.

[SBJ21] Pavel Smirnov, Jeremias Berg, and Matti Järvisalo. Pseudo-Boolean optimization by implicit hitting sets. In *Proceedings of the 27th International Conference on Principles and Practice of Constraint Programming (CP '21)*, volume 210 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 51:1–51:20, October 2021.

[SBJ22] Pavel Smirnov, Jeremias Berg, and Matti Järvisalo. Improvements to the implicit hitting set approach to pseudo-Boolean optimization. In *Proceedings of the 25th International Conference on Theory and Applications of Satisfiability Testing (SAT '22)*, volume 236 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 13:1–13:18, August 2022.

[Sin05] Carsten Sinz. Towards an optimal CNF encoding of Boolean cardinality constraints. In *Proceedings of the 11th International Conference on Principles and Practice of Constraint Programming (CP '05)*, volume 3709 of *Lecture Notes in Computer Science*, pages 827–831. Springer, October 2005.

[SN15] Masahiko Sakai and Hidetomo Nabeshima. Construction of an ROBDD for a PB-constraint in band form and related techniques for PB-solvers. *IEICE Transactions on Information and Systems*, 98-D(6):1121–1127, June 2015.

[Van08] Allen Van Gelder. Verifying RUP proofs of propositional unsatisfiability. In *10th International Symposium on Artificial Intelligence and Mathematics (ISAIM '08)*, 2008. Available at `http://isaim2008.unl.edu/index.php?page=proceedings`.

[VWB22] Dieter Vandesande, Wolf De Wulf, and Bart Bogaerts. Qmaxsatpb: A certified maxsat solver. In Georg Gottlob, Daniela Inclezan, and Marco Maratea, editors, *Logic Programming and Nonmonotonic Reasoning - 16th International Conference, LPNMR 2022, Genova, Italy, September 5-9, 2022, Proceedings*, volume 13416 of *Lecture Notes in Computer Science*, pages 429–442. Springer, 2022.

[War98] Joost P. Warners. A linear-time transformation of linear inequalities into conjunctive normal form. *Information Processing Letters*, 68(2):63–69, 1998.

[WHH14] Nathan Wetzler, Marijn J. H. Heule, and Warren A. Hunt Jr. DRAT-trim: Efficient checking and trimming using expressive clausal proofs. In *Proceedings of the 17th International Conference on Theory and Applications of Satisfiability Testing (SAT '14)*, volume 8561 of *Lecture Notes in Computer Science*, pages 422–429. Springer, July 2014.

Certified Core-Guided MaxSAT Solving

Paper II

Abstract

In the last couple of decades, developments in SAT-based optimization have led to highly efficient maximum satisfiability (MaxSAT) solvers, but in contrast to the SAT solvers on which MaxSAT solving rests, there has been little parallel development of techniques to prove the correctness of MaxSAT results. We show how pseudo-Boolean proof logging can be used to certify state-of-the-art core-guided MaxSAT solving, including advanced techniques like structure sharing, weight-aware core extraction and hardening. Our experimental evaluation demonstrates that this approach is viable in practice. We are hopeful that this is the first step towards general proof logging techniques for MaxSAT solvers.

1 Introduction

Combinatorial optimization is one of the most impressive, and most intriguing, success stories in computer science. This area deals with computationally very challenging problems, which are widely believed to require exponential time in the worst case [IP01, CIP09]. In spite of this, during the last couple of decades astonishing progress has been made on so-called combinatorial solvers for a number of different algorithmic paradigms such as Boolean satisfiability (SAT) solving and optimization [BHvMW21], constraint programming (CP) [RvBW06], and mixed integer programming (MIP) [AW13, BR07]. Today, such solvers are routinely used to solve real-world problems with hundreds of thousands or even millions of variables.

While the performance of modern combinatorial solvers is truly impressive, one negative aspect is that they are highly complex pieces of software, and it is well documented that even mature state-of-the-art solvers sometimes give wrong results [CKSW13, AGJ+18, GSD19, BMN22]. This can be fatal for applications where correctness is a non-negotiable demand. Perhaps the most successful approach for addressing this problem so far is the requirement in the SAT solving

Jeremias Berg, Bart Bogaerts, Jakob Nordström, Andy Oertel, and Dieter Vandesande. "Certified Core-Guided MaxSAT Solving". In *Proceedings of the 29th International Conference on Automated Deduction (CADE-29)*, volume 14132 of *Lecture Notes in Computer Science*, pages 1–22. Springer, July 2023.

community that solvers should be *certifying* [ABM+11, MMNS11], meaning that when given a formula a solver should output not only a verdict whether the formula is satisfiable or unsatisfiable, but also an efficiently machine-verifiable *proof log* establishing that this verdict is guaranteed to be correct. One can then feed the input formula, the verdict, and the proof log to a special, dedicated *proof checker*, and accept the result if the proof checker agrees that the proof log shows that the solver computation is valid. Over the years, different proof formats such as *RUP* [GN03], *TraceCheck* [Bie06], *DRAT* [HHW13a, HHW13b], *GRIT* [CMS17], and *LRAT* [CHH+17] have been developed, and for almost a decade *DRAT* proof logging has been compulsory in the (main track of the) SAT competition. However, there has been very limited progress in designing analogous proof logging techniques for more powerful algorithmic paradigms.

Our focus in this work is on the optimization paradigm that is arguably closest to SAT solving, namely *maximum satisfiability* or *MaxSAT* solving [BJM21, LM21], and the challenge of developing proof logging techniques for MaxSAT solvers.

1.1 Previous Work

Since essentially all modern MaxSAT solvers are based on repeated invocations of SAT solvers, a first question is why SAT proof logging techniques are not sufficient. While *DRAT* is a very powerful proof system, it seems that the overhead of generating proofs of correctness for the rewriting steps in between SAT solver calls in MaxSAT solvers is too large to be tolerable for practical purposes. Another, related, problem is that for optimization problems one needs to reason about the objective function, which *DRAT* struggles to do since its language is limited to disjunctive clauses. But perhaps the biggest challenge is that while modern SAT solving is completely dominated by the *conflict-driven clause learning (CDCL)* method [BS97, MS99, MMZ+01], for MaxSAT there is a rich variety of approaches including *linear SAT-UNSAT* (or *model-improving search*) [ES06, LP10, PRB18], *core-guided search* [FM06, NB14, ADR15, AG17], *implicit hitting set (IHS)* search [DB13a, DB13b], and some recent work on branch-and-bound methods [LXC+22] (where we stress that the lists of references are far from exhaustive).

One tempting solution to circumvent this heterogeneity of solving approaches is to treat the MaxSAT solver as a black box and use a single call to a certifying SAT solver to prove optimality of the final solution found. However, there are several problems with this proposal. Firstly, we would still need proof logging to ensure that the input to the SAT solver is a correct encoding of a claim of optimality for the correct problem instance. Secondly, such a SAT call could be extremely expensive, running counter to the goal of proof logging with low (and predictable) overhead. Finally, even if the SAT-call approach could be made to work efficiently, this would just certify the final result, and would not help validate the correctness of the reasoning of the solver. For these reasons, our goal is to provide proof logging for the actual computations of the MaxSAT algorithm.

While some proof systems and tools have been developed specifically for MaxSAT [BLM07, LNOR11, MM11, MIB+19, FMSV20, PCH20, PCH21, PCH22, IBJ22], none of them comes close to providing general-purpose proof logging,

because they apply only for very specific algorithm implementations and/or fail to capture the full range of reasoning used in an algorithmic approach. A recent work [VDB22] by two co-authors on the current paper instead leverages the pseudo-Boolean proof logging system VeriPB [Ver] to certify correctness of the unweighted linear SAT-UNSAT solver QMaxSAT. VeriPB is similar in spirit to *DRAT*, but operates with more general 0–1 linear inequalities rather than just clauses. This simplifies reasoning about optimization problems, and also makes it possible to capture the powerful MaxSAT solver inferences in a more concise way. VeriPB has previously been used for proof logging of enhanced SAT solving techniques [GN21, BGMN22] and pseudo-Boolean solving [GMNO22], as well as for providing proof-of-concept tools for a nontrivial range of techniques in constraint programming [EGMN20, GMN22] and subgraph solving [GMN20, GMM+20].

1.2 Our Contributions

In this work, we use VeriPB to provide, to the best of our knowledge for the first time, efficient proof logging for the full range of techniques in a cutting-edge MaxSAT solver. We consider the state-of-the-art core-guided solver CGSS [IBJ21], based on RC2 [IMM19], and show how to enhance CGSS to output proofs of correctness of its reasoning, including sophisticated techniques such as stratification [ABGL12, MAGL11], intrinsic-at-most-one constraints [IMM19], hardening [ABGL12], weight-aware core-extraction [BJ17], and structure sharing [IBJ21]. We find that the overhead for such proof logging is perfectly manageable, and although there is certainly room to improve the proof verification time, our experiments demonstrate that already a first proof-of-concept implementation of this approach is practically feasible.

It has been shown previously [EG21, GMM+20, KM21] that proof logging can also serve as a powerful debugging tool. This is because faulty reasoning is likely to lead to unsound proofs, which can be detected even if the solver produces correct output for all test cases. We exhibit yet another example of this—some proofs for which we struggled to make the verification work turned out to reveal two well-hidden bugs in RC2 and CGSS that earlier extensive testing had failed to uncover.

Although it still remains to provide proof logging for other MaxSAT approaches such as (general, weighted) linear SAT-UNSAT and implicit hitting set (IHS) search, we are optimistic that our work could serve as an important step towards general adoption of proof logging techniques for MaxSAT solvers.

1.3 Outline of This Paper

After reviewing preliminaries for pseudo-Boolean reasoning and core-guided MaxSAT solving in Sections 2 and 3, we explain how core-guided MaxSAT solvers can be equipped with proof logging methods in Section 4. In Section 5 we present our experimental evaluation, after which some concluding remarks and directions for future research are given in Section 6.

2 Preliminaries

We start by a review of some standard material which can be found, e.g., in [BN21, GN21, GMNO22]. A *literal* ℓ over a Boolean variable x (taking values in $\{0,1\}$, which we identify with false and true, respectively) is x itself or its negation $\overline{x}$, where $\overline{x} = 1 - x$. A *pseudo-Boolean (PB)* constraint is a 0-1 integer linear inequality $C \doteq \sum_i a_i \ell_i \geq A$ (where $\doteq$ denotes syntactic equality). When convenient, we can assume without loss of generality that PB constraints are in *normalized form* [Bar95]; i.e., all literals ℓ_i are over distinct variables and the coefficients a_i and the *degree (of falsity)* A are non-negative integers. The set of literals in C is denoted $lits(C)$. The *negation* of C is $\neg C \doteq \sum_i a_i \ell_i \leq A - 1$ (rewritten in normalized form when needed). A *pseudo-Boolean formula* is a conjunction $F \doteq \bigwedge_j C_j$ of PB constraints. Note that a disjunctive clause can be viewed as a PB constraint with all coefficients and the degree equal to 1, and so formulas in conjunctive normal form (CNF) are special cases of PB formulas.

A *(partial) assignment* ρ is a (partial) function from variables to $\{0,1\}$, which we extend to literals by respecting the meaning of negation. Applying ρ to a constraint C yields $C{\upharpoonright}_\rho$ by substituting the variables assigned in ρ by their values, and for a formula $F \doteq \bigwedge_j C_j$ we define $F{\upharpoonright}_\rho \doteq \bigwedge_j C_j{\upharpoonright}_\rho$. The constraint C is *satisfied* by ρ if $\sum_{\rho(\ell_i)=1} a_i \geq A$, and ρ satisfies F if it satisfies all $C \in F$, in which case F is *satisfiable*. A formula lacking satisfying assignments is *unsatisfiable*. We say that F *implies* C, denoted $F \models C$, if any assignment satisfying F also satisfies C.

An *objective* $O \doteq \sum_i w_i \ell_i + M$ is an affine function over literals ℓ_i to be minimized by (total) assignments α satisfying F. The *value* (or *cost*) of an objective O under such an α, which we refer to as a *solution*, is $O(\alpha) = \sum_{\alpha(\ell_i)=1} w_i + M$. We write $\mathit{coeff}(O, \ell_i)$ to denote the coefficient w_i of a literal $\ell_i \in lits(O)$.

The foundation of the pseudo-Boolean proof logging in this paper is the *cutting planes* proof system [CCT87], which is a method to iteratively derive new constraints implied by a pseudo-Boolean formula F. If C and D have been derived before or are *axiom constraints* in F, then any positive *linear combination* of these constraints can be derived. *Literal axioms* $\ell \geq 0$ can also be added to any previously derived constraints. For a constraint $\sum_i a_i \ell_i \geq A$ in normalized form, *division* by a positive integer d derives $\sum_i \lceil a_i/d \rceil \ell_i \geq \lceil A/d \rceil$, and we also add a *saturation* rule that derives $\sum_i \min\{a_i, A\} \cdot \ell_i \geq A$ (where the soundness of these rules crucially depends on the normalized form). It is well known that any PB constraint implied by F can be derived using these rules.

A constraint C is said to *unit propagate* the literal ℓ to true under an assignment ρ if $C{\upharpoonright}_\rho$ cannot be satisfied unless ℓ is true. During *unit propagation* on F under ρ, we extend ρ iteratively by any propagated literals until an assignment ρ' is reached under which no constraint $C \in F$ is propagating or some constraint C wants to propagate a literal that has already been assigned to the opposite value. The latter case is called a *conflict*, since C is *violated* by ρ'. We say that F implies C by *reverse unit propagation (RUP)*, and that C is a *RUP constraint* with respect to F, if $F \wedge \neg C$ unit propagates to conflict under the empty assignment. It is not hard to see that $F \models C$ holds if C is a RUP constraint, and as a convenient shorthand we will add a RUP rule for deriving new constraints.

In addition to deriving constraints that are implied by a formula F, we also allow deriving so-called *redundant* constraints C that are *not* implied by F as long as some optimal solution is guaranteed to be preserved. This is done by extending the proof system with a *redundance-based strengthening* rule [GN21, BGMN22]. We will only need the special case of this rule saying that for a fresh variable z and for any constraint $D \doteq \sum_i a_i \ell_i \geq A$ we can introduce the *reified constraints*

$$C^{\Rightarrow}_{\text{reif}}(z, D) \doteq A\bar{z} + \textstyle\sum_i a_i \ell_i \geq A \tag{1a}$$

$$C^{\Leftarrow}_{\text{reif}}(z, D) \doteq (\textstyle\sum_i a_i - A + 1)\, z + \sum_i a_i \bar{\ell}_i \geq \sum_i a_i - A + 1 \tag{1b}$$

encoding the implications $z \Rightarrow D$ and $z \Leftarrow D$, respectively. We refer to z as the *reification variable*, and when D is clear from context, we will sometimes write just $C^{\Rightarrow}_{\text{reif}}(z)$ for (1a) and $C^{\Leftarrow}_{\text{reif}}(z)$ for (1b).

The *maximum satisfiability (MaxSAT) problem* can be described conveniently as a special case of pseudo-Boolean optimization. A discussion on the equivalence of the following and the—more classical—clause-centric definition can be found in, for instance, [LBJ20, BJM21]. An instance (F, O) of the (weighted partial) MaxSAT problem consists of a CNF formula F and an objective function O written as a non-negative affine combination of literals. The goal is to find a solution α that satisfies F and minimizes $O(\alpha)$. We say that such a solution α is *optimal* for the instance and that the optimal cost of the instance (F, O) is $O(\alpha)$.

3 The OLL Algorithm for Core-Guided MaxSAT Solving

We now proceed to discuss the core-guided MaxSAT solving in CGSS, which is based on the OLL algorithm [AKMS12, MDM14], and describe the main heuristics used in efficient implementations of this algorithm. Given a MaxSAT instance (F_{orig}, O_{orig}), OLL takes an optimistic view and attempts to find an assignment satisfying F_{orig} in which O_{orig} equals its constant term (i.e., all literals in $lits(O_{orig})$ are false). If such a solution exists, it is clearly optimal. Otherwise, the solver will extract a *core* K, which is a clause such that (i) K only contains objective literals, i.e., $lits(K) \subseteq lits(O_{orig})$, and (ii) F_{orig} implies K, which means that any solution to F_{orig} has to set at least one literal in $lits(K)$ to true. The *cost* $w(K, O) = \min\{coeff(O, \ell) : \ell \in lits(K)\}$ of a core K is the smallest coefficient in the objective O of any literal in K. The core K is used to (conceptually) reformulate the instance into (F_{ref}, O_{ref}) which has the same minimal-cost solutions. The constant term LB in O_{ref} is a lower bound on the optimal cost of the instance, and the reformulation is done in such a way that the lower bound increases (exactly) with the cost of the core K as defined above.

In more detail, the algorithm maintains a reformulated objective O_{ref} (initialized to O_{orig}) such that the (non-normalized) pseudo-Boolean constraint

$$O_{orig} \geq O_{ref} \doteq \sum_{b \in lits(O_{orig})} coeff(O_{orig}, b) \cdot b \geq \sum_{b' \in lits(O_{ref})} coeff(O_{ref}, b') \cdot b' + LB \tag{2}$$

is satisfied by all solutions of F_{ref}. Note that the constraint (2), which we refer to as an *objective reformulation constraint*, implies that the constant term LB is a lower bound on the optimal cost.

In each iteration, a SAT solver is queried for a solution α to F_{ref} with $O_{ref}(\alpha) = LB$. If such an α exists, the constraint (2) yields that $O_{orig}(\alpha) = LB$, and so α is a minimal-cost solution to (F_{orig}, O_{orig}). Otherwise, the solver returns a new core K that requires at least one literal in $lits(O_{ref})$ to be set to 1. This implies that the optimal cost is strictly larger than LB, and the core K is used for a new reformulation step.

The objective reformulation step adds new clauses to F_{ref} encoding the constraints $y_{K,k} \Leftarrow \sum_{b \in lits(K)} b \geq k$ for $k = 2, \ldots, |K|$. The new variables $y_{K,k}$ are added to O_{ref} with coefficient $w(K, O_{ref})$ equalling the cost of K, and the coefficient in O_{ref} of each literal in K is decreased by the same amount. Finally, the lower bound LB—the constant term of O_{ref}—is also increased by $w(K, O_{ref})$. Since $y_{K,k}$ encodes that at least k literals in K are true, we have the equality $\sum_{b \in lits(K)} b = 1 + \sum_{k=2}^{|K|} y_{K,k}$, where the additive 1 comes from the fact that at least one literal in K has to be true, and the reformulation step is just applying this equality multiplied by $w(K, O_{ref})$ to O_{ref}. Notice that the variables added during objective reformulation can later be discovered in other cores. In practice, all implementations of OLL we are aware of encode the semantics of counting variables incrementally [MJML14]. This means that initially only the variable $y_{K,2}$ is defined, and the variable $y_{K,i+1}$ is introduced only after $y_{K,i}$ is found in a core.

Implementations of OLL for MaxSAT—including the CGSS solver that we enhance with proof logging in this work—extend the algorithm with a number of heuristics such as stratification [ABGL12, MAGL11], hardening [ABGL12], the intrinsic-at-most-ones technique [IMM19], weight-aware core extraction [BJ17], and structure sharing [IBJ21].

Stratification extracts cores not over all literals in O_{ref} but only over those whose coefficient is above some bound w_{strat}. This steers search toward cores containing literals with high coefficients, resulting in larger increases of LB. Once no more cores over such variables can be found, the algorithm lowers w_{strat}, terminating only after no more cores can be found with $w_{strat} = 1$. The fact that no more cores containing only variables with coefficients above w_{strat} exist is detected by the SAT solver returning a (possibly non-optimal) solution α. The minimal cost $O_{orig}(\alpha)$ of all such solutions gives an upper bound UB on the optimal cost of the instance, allowing OLL to terminate as soon as $LB = UB$.

Hardening fixes literals in O_{ref} to 0 based on information provided by the current upper and lower bounds UB and LB. If for any $b \in lits(O_{ref})$ it holds that $coeff(O_{ref}, b) + LB > UB$, then any solution α with $b = 1$ would have higher cost than the current best solution known, and would thus not be optimal.

The *intrinsic-at-most-one* technique identifies subsets $\mathcal{S} \subseteq lits(O_{ref})$ of objective literals such that $\sum_{b \in \mathcal{S}} \overline{b} \leq 1$ is implied, i.e., any solution can assign at most one literal in $\mathcal{S}$ to 0. This is used both to increase the lower bound and to reformulate the objective. If we let $w_{min} = \min\{coeff(O_{ref}, b) : b \in \mathcal{S}\}$, then $\mathcal{S}$ implies a lower bound increase of $LB_{\mathcal{S}} = (|\mathcal{S}| - 1) \cdot w_{min}$. Additionally, we define a new variable $\ell_{\mathcal{S}}$ by the clause $\ell_{\mathcal{S}} + \sum_{b \in \mathcal{S}} \overline{b} \geq 1$ to indicate if in fact all literals in $\mathcal{S}$ are true, and introduce it in the reformulated objective with coefficient w_{min}. This means that we

remove the already known lower bound $LB_{\mathcal{S}}$ from O_{ref} and transfer the possible additional cost w_{min} from $\mathcal{S}$ to the variable $\ell_{\mathcal{S}}$.

Weight-aware core extraction (WCE) delays objective reformulation, and the accompanying increase in new variables and clauses, for as long as possible. When a new core K is extracted by a solver that uses WCE, initially only the coefficient of each $b \in lits(K)$ is lowered and the lower bound LB is increased by $w(K, O_{ref})$. Then the SAT solver is invoked again with the literals, that still have coefficients above w_{strat} in O_{ref}, set to 0. When the SAT solver finds a satisfying assignment extending the assumptions, all objective reformulations steps are then performed at once. This is correct since the final effect is the same as if the core would have been discovered one by one and immediately followed by objective reformulation. Notice that this core extraction loop is guaranteed to terminate since the coefficient of at least one variable is decreased to 0 for each new core. *Structure sharing* is a recent extension to weight-aware core extraction that makes use of the potential overlap in cores detected in order to achieve more compact encodings of counting variable semantics.

4 Proof Logging for the OLL Algorithm for MaxSAT

We have now reached a point where we can describe the contribution of this work, namely how to add proof logging to an OLL-based core-guided MaxSAT solver, including all the state-of-the-art techniques described in Section 3.

In our proof logging routines we maintain the invariants described next. The reformulated objective O_{ref} is already implicitly tracked by the solver and at all times it is possible to derive that $O_{orig} \geq O_{ref}$ as in (2). We also keep track of the current upper bound UB on O_{orig} and best solution α_{best} found so far. All cores that have been found and processed are in the set $\mathcal{K}$.

SAT Solver Calls. The CDCL SAT solvers used in core-guided MaxSAT algorithms can support *DRAT* proof logging, and since the proof format used by VeriPB is a strict extension of *DRAT* (modulo small and purely syntactical modifications) it is straightforward to provide proof logging for the part of the reasoning done in SAT solver calls, and to add all learned clauses to the proof checker database.

Each invocation of the SAT solver returns either a new solution α or a new core K. When a solution α with $O_{orig}(\alpha) < UB$ is obtained, it is logged in the proof, which adds the *objective-improving constraint*

$$O_{orig} \leq UB - 1 \tag{3a}$$

(which is

$$\sum_{b \in lits(O_{orig})} coeff(O_{orig}, b) \cdot \overline{b} \;\geq\; 1 + \sum_{b \in lits(O_{orig})} coeff(O_{orig}, b) - UB \tag{3b}$$

in normalized form). A technical side remark is that later solutions with cost greater than UB cannot successfully be logged, since they violate the constraint (3a) added to the proof checker database, and so the proof logging routines make sure to only log solutions that improve the current upper bound.

If the SAT solver instead returns a new core K, this clause is guaranteed to be a reverse unit propagation (RUP) clause with respect to the set of clauses currently in the solver database, and so we can use the RUP rule to add K to the proof checker database (which contains a superset of the clauses known by the solver). For our book-keeping, we also add K to the set $\mathcal{K}$. A special case is that K could be the contradictory empty clause, corresponding to the pseudo-Boolean constraint $0 \geq 1$. This means that there are no solutions to the formula.

To optimize the efficiency of proof verification, constraints should be deleted from the proof when they are no longer needed. Since SAT solver proofs are only used to prove *unsatisfiability* this does not cause any issues, but when certifying *optimality* we have to be careful in order not to create better-than-optimal solutions (which could happen if, e.g., constraints in the input formula are removed). The *checked deletion* rule [BGMN22] ensuring this in VeriPB does not have any analogue in *DRAT*, so some care is needed here when translating SAT solver proofs into the VeriPB format.

Incremental Totalizer with Structure Sharing. Different implementations of OLL for MaxSAT differ in which encoding is used for the counting variables introduced during objective reformulation [KP18, KP19, BB03]. The two solvers we consider use totalizers [BB03], so we start by explaining this encoding and then show how to provide proof logging for the clauses added to the proof checker database.

The totalizer encoding for a set $I = \{\ell_1, \ldots, \ell_n\}$ of literals is a CNF formula $\mathcal{T}$ that defines *counting variables* $y_{I,j}$ for $j = 1, \ldots, n$ such that for any assignment that satisfies $\mathcal{T}$ the variable $y_{I,j}$ is true if and only if $\sum_{i=1}^{n} \ell_i \geq j$. The structure of $\mathcal{T}$ can be viewed as a binary tree, with literals in I at the leaves and with each internal node η associated with variables counting the true leaf literals in the subtree rooted at η. The variables $y_{I,j}$ are associated with the root of the tree.

More formally, given a set of literals I, we construct a binary tree with leaves labelled by the literals in I. For every node η of $\mathcal{T}$, let $lits(\eta)$ denote the leaves in the subtree rooted at η; where it is convenient, we will overload I to also refer to the root note. For each internal node η, the totalizer encoding introduces the counting variables $S_\eta = \{y_{\eta,1}, \ldots, y_{\eta,|lits(\eta)|}\}$, the meaning of which can be encoded recursively in terms of the variables S_{η_1} and S_{η_2} for the children η_1 and η_2 of η by the (pseudo-Boolean form of the) clauses

$$C_\eta^{\Leftarrow}(\alpha, \beta, \sigma) \doteq y_{\eta,\sigma} + \overline{y}_{\eta_1,\alpha} + \overline{y}_{\eta_2,\beta} \geq 1 \tag{4a}$$

$$C_\eta^{\Rightarrow}(\alpha, \beta, \sigma) \doteq \overline{y}_{\eta,\sigma+1} + y_{\eta_1,\alpha+1} + y_{\eta_2,\beta+1} \geq 1 \tag{4b}$$

for all integers α, β, σ such that $\alpha + \beta = \sigma$ and $0 \leq \alpha \leq |lits(\eta_1)|$, $0 \leq \beta \leq |lits(\eta_2)|$, and $0 \leq \sigma \leq |lits(\eta)|$. We use the notational conventions in (4a)–(4b) that $y_{\ell,1} = \ell$ for all leaves ℓ, and that $y_{\eta,0} = 1$ and $y_{\eta,|lits(\eta)|+1} = 0$ for all nodes η (so that clauses containing $y_{\eta,0}$ or $y_{\eta,|lits(\eta)|+1}$ can be simplified to binary clauses or be omitted when they are satisfied). The clauses $C_\eta^{\Rightarrow}(\alpha, \beta, \sigma)$ in (4b) are not necessarily added to the clause database of the MaxSAT solver, but are sometimes included for improved propagation.

We now turn to the question of how to derive the clauses (4a)–(4b) encoding the meaning of the counting variables $y_{I,j}$ in the proof. This is a two-step process. First, reified pseudo-Boolean (and, in general, non-clausal) constraints $C^{\Rightarrow}_{\mathrm{reif}}(y_{\eta,j})$ and $C^{\Leftarrow}_{\mathrm{reif}}(y_{\eta,j})$ as in (1a)–(1b), encoding that $y_{\eta,j}$ holds if and only if $\sum_{\ell \in lits(\eta)} \ell \geq j$, are derived by redundance-based strengthening. Then the clauses added to the MaxSAT solver are derived from these pseudo-Boolean constraints. Although we omit the details due to space constraints, it is not hard to show that for any internal node η with children η_1 and η_2, the clauses $C^{\Leftarrow}_{\eta}(\alpha, \beta, \sigma)$ and $C^{\Rightarrow}_{\eta}(\alpha, \beta, \sigma)$ in (4a)–(4b) can be derived from the constraints $C^{\Leftarrow}_{\mathrm{reif}}(y_{\eta,\sigma})$, $C^{\Rightarrow}_{\mathrm{reif}}(y_{\eta,\sigma})$, $C^{\Leftarrow}_{\mathrm{reif}}(y_{\eta_1,\alpha})$, $C^{\Rightarrow}_{\mathrm{reif}}(y_{\eta_1,\alpha})$, $C^{\Leftarrow}_{\mathrm{reif}}(y_{\eta_2,\beta})$, and $C^{\Rightarrow}_{\mathrm{reif}}(y_{\eta_2,\beta})$ by standard cutting planes derivations as in [VDB22]. In particular, the certification of these totalizers can be done incrementally: clauses in the encoding can be derived as the corresponding counter variables are lazily introduced in the OLL algorithm.

This approach is also compatible with structure sharing, where subtrees of a previously constructed totalizer tree can be reused (to avoid doing the same work twice). The only constraints from a subtree rooted at η^* that are needed when generating another totalizer encoding at a higher level are the constraints $C^{\Rightarrow}_{\mathrm{reif}}(y_{\eta^*,\sigma})$ and $C^{\Leftarrow}_{\mathrm{reif}}(y_{\eta^*,\sigma})$ defining the counter variables in the subtree root η^*.

To decrease the memory usage of the proof checker, it can be useful to *delete* reification constraints from the proof once we know that they will no longer be needed. Without structure sharing, for an internal node η, once all clauses that mention $y_{\eta,j}$ are created, the constraints $C^{\Leftarrow}_{\mathrm{reif}}(y_{\eta,j})$ and $C^{\Rightarrow}_{\mathrm{reif}}(y_{\eta,j})$ will not be used anymore and can thus be deleted. On the other hand, structure sharing reuses as many counting variables as possible, even over multiple iterations of weight-aware core extraction. This means that $C^{\Leftarrow}_{\mathrm{reif}}(y_{\eta,j})$ and $C^{\Rightarrow}_{\mathrm{reif}}(y_{\eta,j})$ need to be retained, even after all clauses in the totalizer encoding for all parents of node η have been created.

Objective Reformulation. If counting variables $y_{K,i}$ for $i = 2, \ldots, s_K$ have been introduced for the core K, then the objective reformulation with respect to K is derived with the help of the constraint

$$\sum_{b \in K} b \geq 1 + \sum_{i=2}^{s_K} y_{K,i} \tag{5a}$$

(or

$$\sum_{b \in K} b + \sum_{i=2}^{s_K} \overline{y}_{K,i} \geq s_K \tag{5b}$$

in normalized form). The constraint (5b) can in turn be obtained from the core clause K and the reified constraints $C^{\Rightarrow}_{\mathrm{reif}}(y_{K,j})$. It is clear that this should be possible, since the latter constraints define the variables $y_{K,j}$ precisely so that (5b) should hold, and we refer to Algorithm 5 in [GMNO22] for the details. Also, each time a new counting variable $y_{K,j}$ is introduced for a core K, we add it to (5b) to maintain this constraint as an invariant.

To illustrate how this update works, suppose we have a core $K \doteq \sum_{i=1}^{n} b_i \geq 1$ for which $\sum_{i=1}^{n} b + \sum_{i=2}^{s_K-1} \overline{y}_{K,i} \geq s_K - 1$ has already been derived. The next counting

variable y_{K,s_K} is introduced by the reification $s_K \cdot \overline{y}_{K,s_K} + \sum_{i=1}^{n} b_i \geq s_K$. The previous constraint is multiplied by $s_K - 1$ and added to the new reified constraint, yielding $s_K \cdot \sum_{i=1}^{n} b + (s_K - 1) \cdot \sum_{i=2}^{s_K - 1} \overline{y}_{K,i} + s_K \cdot \overline{y}_{K,s_K} \geq (s_K - 1) \cdot s_K + 1$. Dividing this last constraint by s_K results in $\sum_{i=1}^{n} b + \sum_{i=2}^{s_K} \overline{y}_{K,i} \geq s_K$, which is the desired updated constraint.

For a set of extracted cores $\mathcal{K}$, we can derive the objective reformulation constraint $O_{orig} \geq O_{ref}$ by multiplying (5b) for each $K \in \mathcal{K}$ by the cost $w(K, O_{ref})$ of K and summing up all these multiplied constraints. The fact that we have an inequality $O_{orig} \geq O_{ref}$ rather than an equality is due to the incremental use of totalizers. More specifically, if $s_K = |lits(K)|$ would hold for every $K \in \mathcal{K}$, it would be possible to derive $O_{orig} = O_{ref}$ instead. Here we would like to stress one subtlety for developing proof logging for OLL: as the algorithm progresses and more output variables of totalizers are introduced (i.e., the counters s_K increase), the reformulated objective potentially also increases—because of added counted variables when s_K increases we have the inequality $O_{orig} \geq O_{ref}^{new} \geq O_{ref}^{old}$. For this reason, the old constraint $O_{orig} \geq O_{ref}^{old}$ cannot be used to derive $O_{orig} \geq O_{ref}^{new}$ after objective reformulation. Instead, we have to derive $O_{orig} \geq O_{ref}$ from scratch each time the solver argues with the reformulated objective. For doing this we need to have access to the entire set $\mathcal{K}$ of cores.

Proving Optimality. When the solver has found an optimal solution and established a matching lower bound, optimality is certified in the proof log using a proof by contradiction from the objective reformulation constraint $O_{orig} \geq O_{ref}$ in (2) and the (normalized form of the) objective-improving constraint $O_{orig} \leq UB - 1$ in (3b). If we add these two constraints and cancel like terms, we get

$$\sum_{b' \in lits(O_{ref})} coeff(O_{ref}, b') \cdot \overline{b'} \geq 1 - UB + LB + \sum_{b' \in lits(O_{ref})} coeff(O_{ref}, b') \,. \tag{6}$$

Since we have $UB = LB$ when the optimal solution has been found, and since $\sum_{b' \in lits(O_{ref})} coeff(O_{ref}, b') \cdot \overline{b'}$ cannot possibly exceed $\sum_{b' \in lits(O_{ref})} coeff(O_{ref}, b')$, the constraint (6) can be simplified to contradiction $0 \geq 1$.

Intrinsic At-Most-One Constraints. Certifying intrinsic at-most-one constraints for a set $\mathcal{S} \subseteq lits(O_{ref})$ of literals requires deriving (i) the at-most-one constraint stating that at most one $b \in \mathcal{S}$ is assigned to 0 by any solution and (ii) constraints defining the variable $\ell_{\mathcal{S}}$. Such sets $\mathcal{S}$ are detected by unit propagation that implicitly derives implications $\overline{b}_i \Rightarrow b_j$ in the form of binary clauses $b_i + b_j \geq 1$ for every pair of variables in $\mathcal{S}$. In the proof log, all these binary clauses can be obtained by RUP steps, after which the at-most-one constraint $\sum_{b \in \mathcal{S}} \overline{b} \leq 1$ (which is $\sum_{b \in \mathcal{S}} b \geq |\mathcal{S}| - 1$ in normalized form) is derived by a standard cutting planes derivation (see, e.g., [CCT87]).

The reified constraints $\ell_{\mathcal{S}} \Leftarrow \sum_{b \in \mathcal{S}} b \geq |S|$ and $\ell_{\mathcal{S}} \Rightarrow \sum_{b \in \mathcal{S}} b \geq |S|$ defining the variable ℓ_S (which are $\ell_{\mathcal{S}} + \sum_{b \in \mathcal{S}} \overline{b} \geq 1$ and $\overline{\ell}_{\mathcal{S}} + \sum_{b \in \mathcal{S}} b \geq |\mathcal{S}|$, respectively, in normalized form) are derived by redundance-based strengthening. Note that the

Table 1: *Example proof produced by a certified OLL solver.*

id	Pseudo-Boolean constraint	Justification
(1)	$b_1 + x \geq 1$	input
(2)	$b_2 + \overline{x} \geq 1$	input
(3)	$b_3 + b_4 \geq 1$	input
(4)	$5\overline{b}_1 + 5\overline{b}_2 + \overline{b}_3 + \overline{b}_4 \geq 6$	log solution α_1
(5)	$b_1 + b_2 \geq 1$	RUP
(6)	$\overline{b}_1 + \overline{b}_2 + y_{K_1,2} \geq 1$	reification
(7)	$2\overline{y}_{K_1,2} + b_1 + b_2 \geq 2$	reification
(8)	$5b_1 + 5b_2 + 5\overline{y}_{K_1,2} \geq 10$	$(((5) + (7))/2) \cdot 5$
(9)	$\overline{b}_3 + \overline{b}_4 + 5\overline{y}_{K_1,2} \geq 6$	(4) + (8)
(10)	$\overline{y}_{K_1,2} \geq 1$	RUP
(11)	$b_3 + b_4 \geq 1$	RUP
(12)	$\overline{b}_3 + \overline{b}_4 + y_{K_2,2} \geq 1$	reification
(13)	$2\overline{y}_{K_2,2} + b_3 + b_4 \geq 2$	reification
(14)	$b_3 + b_4 + \overline{y}_{K_2,2} \geq 2$	((11) + (13))/2
(15)	$5\overline{b}_1 + 5\overline{b}_2 + \overline{b}_3 + \overline{b}_4 \geq 7$	log solution α_2
(16)	$5b_1 + 5b_2 + b_3 + b_4 + 5\overline{y}_{K_1,2} + \overline{y}_{K_2,2} \geq 12$	(8) + (14)
(17)	$5\overline{y}_{K_1,2} + \overline{y}_{K_2,2} \geq 7$	(15) + (16), $\perp$

latter constraint does not exist in the MaxSAT solver, but we need it in the proof in order to derive the objective reformulation for the at-most-one constraint.

Hardening. Formally, hardening corresponds to deriving $\overline{b} \geq 1$ in the proof for some literal $b \in lits(O_{ref})$ for which $UB < LB + coeff(O_{ref}, b)$ holds. Such an inequality $\overline{b} \geq 1$ is implied by RUP if we first derive the constraint (6), since assigning $b = 1$ results in (6) being contradicting.

Upper Bound Estimation. A final technical proof logging detail is that some implementations of the OLL algorithm for MaxSAT—including the Python-based version of CGSS—do not use the actual cost of the solution found by the SAT solver as the upper bound UB when hardening. In order to avoid the overhead in Python of extracting the solution from the SAT solver, an upper bound estimate UB_{est} is computed instead based on the initial assignment passed to the SAT solver in the call. Since any valid estimate is at least the cost of the solution found (i.e., $UB_{est} \geq UB$), hardening steps based on UB_{est} can be justified by first deriving $O_{orig} \leq UB_{est} - 1$, which follows from the latest objective-improving constraint (3a). However, in order to handle solutions correctly in the proof, the proof logging routines need to extract the solution found by the solver and compute the actual cost, which means that a Python-based solver will not be able to avoid this overhead when running with proof logging.

Worked-Out Example. We end this section with a complete, worked-out example of OLL solving and proof logging for the toy MaxSAT instance (F, O) with formula $F = \{(b_1 \vee x), (\neg x \vee b_2), (b_3 \vee b_4)\}$ and objective $O = 5b_1 + 5b_2 + b_3 + b_4$.

After initialization, the internal SAT solver of the OLL algorithm is loaded with the clauses of F and the proof consists of constraints (1)–(3) in Table 1. The OLL search begins by invoking the SAT solver on the clauses in F in order to check the existence of any solutions. Assume the SAT solver returns the solution α_1 assigning $b_1 = b_3 = b_4 = 1$ and $b_2 = x = 0$. This solution has objective value $O(\alpha_1) = O_{orig}(\alpha_1) = 7$ so the algorithm updates $UB = 7$ and logs the objective-improving constraint (4) in Table 1 equivalent to $O_{orig} \leq 6$.

Assume the stratification bound w_{strat} is initialised to 2. Then the solver is invoked with $b_1 = b_2 = 0$ and returns the core $K_1 \doteq b_1 + b_2 \geq 1$, which is added to the proof as constraint (5). As already mentioned, core clauses are guaranteed to be RUP with respect to the set of clauses in the SAT solver database, which are also added to the proof.

For simplicity, we ignore WCE and structure sharing in this example, meaning that the solver next reformulates the objective based on K_1 by introducing clauses enforcing $y_{K_1,2} \Leftarrow (b_1 + b_2 \geq 2)$ for the new counting variable $y_{K_1,2}$. This is done by (i) introducing the pseudo-Boolean constraints (6) and (7) in Table 1 by reification, and (ii) deriving the clauses corresponding to these constraints. While the MaxSAT solver only uses the implication (6), the proof also requires constraint (7) corresponding to $y_{K_1,2} \Rightarrow (b_1 + b_2 \geq 2)$. Conveniently, in this toy example $y_{K_1,2} \Leftarrow (b_1 + b_2 \geq 2)$ is already the clause $\overline{b}_1 + \overline{b}_2 + y_{K_1,2} \geq 1$, so step (ii) is not needed. For the general case, we derive totalizer clauses as explained in Section 4. Conceptually, we now replace $5b_1 + 5b_2$ by $5y_{K_1,2} + 5$ to obtain the reformulated objective $O_{ref} = b_3 + b_3 + 5y_{K_1,2} + 5$ with lower bound $LB = 5$. The core K_1 says that at least one of b_1 and b_2 must be true, thus incurring a cost of 5, and $y_{K_1,2}$ is added to the objective to indicate if both of them incur cost.

Since it now holds that $\mathit{coeff}(O_{ref}, y_{K_1,2}) + LB = 5 + 5 \geq 7 = UB$, the literal $y_{K_1,2}$ is hardened to 0. In order to certify this hardening step, i.e., derive $\overline{y}_{K_1,2} \geq 1$, the proof logger first derives the objective reformulation constraint $5b_1 + 5b_2 + b_3 + b_4 \geq b_3 + b_4 + 5y_{K_1,2} + 5$ enforced by line (8) in Table 1. The objective-improving and objective reformulation constraints are then added together to get constraint (9), after which $\overline{y}_{K_1,2} \geq 1$ is obtained by a RUP step.

The next SAT solver call with $b_3 = b_4 = 0$ returns as core the input clause $b_3 + b_4 \geq 1$, and reformulation (lines (11)–(13)) yields $O_{ref} = 5y_{K_1,2} + y_{K_2,2} + 6$ with $LB = 6$. Now suppose the SAT solver finds the solution α_2 with $b_2 = b_3 = x = 1$ and all other variables set to 0, resulting in the objective-improving constraint (15). Since $O_{orig}(\alpha_2) = 6 = LB$, the solver terminates and reports α_2 to be optimal. To certify that this is correct, another objective reformulation constraint (16) is derived, after which the contradictory constraint (17) is obtained by adding (15) and (16). This proves that solutions with cost less than 6 do not exist.

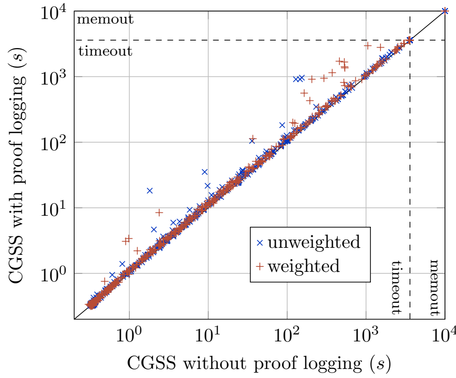


Figure 1: *Running time of CGSS with and without proof logging.*

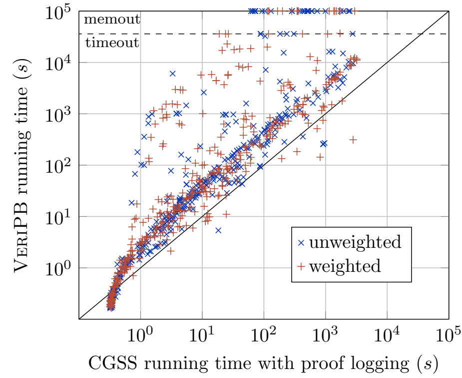


Figure 2: *CGSS running time compared to time required for proof checking.*

5 Experimental Evaluation

To evaluate the proof logging techniques developed in this paper, we have implemented them in the state-of-the-art MaxSAT solver CGSS [CGSa, IBJ21], which uses the OLL algorithm and structure-sharing totalizers. We employed VeriPB [Ver], extended to parse MaxSAT instances in the standard WCNF format, to verify the certificates of correctness emitted by the certifying solver.

Our experiments were conducted on machines with an 11th Gen Intel(R) Core(TM) i5-1145G7 @ 2.60 GHz CPU and 16 GB of memory. Each benchmark ran exclusively on a single machine with a memory limit of 14 GB and a time limit of 3 600 seconds for solving with CGSS and 36 000 seconds for checking the certificates with VeriPB. As benchmarks we used all 594 weighted and 607 unweighted instances from the complete track of the MaxSAT Evaluation 2022 [Max22], where an an instance (F, O) is *unweighted* if all coefficents $\mathit{coeff}(O, \ell)$ are equal. The data from our experiments can be found in [BBN+23].

Overhead of Proof Logging. To evaluate the overhead in solver running time, we compared the standard CGSS solver [CGSb] without proof logging (but with the bug fixes discussed below) to CGSS with proof logging as described in this paper. With proof logging 803 instances are solved within the resource limits, which is 3 instances less than without proof logging (see Figure 1). Adding proof logging slowed down CGSS by about 8.8% in the median over all solved instances. For 95% of the instances CGSS with proof logging was at most 36.2% slower. Thus, the proof logging overhead seems perfectly manageable and should present no serious obstacles to using proof logging in core-guided MaxSAT solvers.

Overhead of Proof Checking. To assess the efficiency of proof checking, we compared the running time of CGSS with proof logging to the time taken by VeriPB for checking the generated proofs. The instances that were not solved

Table 2: *Illustration of discovered bug (where $y_{i,k}$ should be read as $y_{K_i,k}$).*

#iter	Literals considered ($w_{strat} = 2$)	Core $K_{\#\text{iter}}$ extracted
1	$\{b_i, e_i \mid i = 1 \ldots 5\}$	$K_1 = \sum_{i=1}^{5} b_i \geq 1$
2	$\{e_i \mid i = 1 \ldots 5\} \cup \{y_{1,2}\}$	$K_2 = y_{1,2} + e_2 + e_4 \geq 1$
3	$\{e_i \mid i = 1 \ldots 3, 5\} \cup \{y_{1,2}, y_{1,3}\} \cup \{y_{2,2}\}$	$K_3 = y_{1,3} + e_1 + e_2 + e_5 \geq 1$
4	$\{e_i \mid i = 1 \ldots 3\} \cup \{y_{1,2}, y_{1,4}\} \cup \{y_{2,2}, y_{3,2}\}$	$K_4 = y_{1,2} + e_1 + e_2 \geq 1$
5	$\{e_i \mid i = 1 \ldots 3\} \cup \{y_{1,4}\} \cup \{y_{2,2}, y_{3,2}, y_{4,2}\}$	$K_5 = e_1 + e_2 + e_3 + y_{1,4} + y_{2,2} \geq 1$
6	$\{e_3\} \cup \{y_{1,5}\} \cup \{y_{2,3}\} \cup \{y_{3,2}, y_{4,2}, y_{5,2}\}$	Result is SAT

#iter	O_{ref} (after reformulation of $K_{\#\text{iter}}$)
0	$10\left(\sum_{i=1}^{5} b_i\right) + 11e_1 + 14e_2 + 11e_3 + 3e_4 + 2e_5 + o_1 + o_2$
1	$11e_1 + 14e_2 + 11e_3 + 3e_4 + 2e_5 + 10y_{1,2} + o_1 + o_2 + 10$
2	$11e_1 + 11e_2 + 11e_3 + 2e_5 + 7y_{1,2} + 3y_{1,3} + 3y_{2,2} + o_1 + o_2 + 13$
3	$9e_1 + 9e_2 + 11e_3 + 7y_{1,2} + y_{1,3} + 2y_{1,4} + 3y_{2,2} + 2y_{3,2} + o_1 + o_2 + 15$
4	$2e_1 + 2e_2 + 11e_3 + \mathbf{8y_{1,3}} + 2y_{1,4} + 3y_{2,2} + 2y_{3,2} + 7y_{4,2} + o_1 + o_2 + 22$
5	$9e_3 + \mathbf{8y_{1,3}} + 2y_{1,5} + y_{2,2} + 2y_{2,3} + 2y_{3,2} + 7y_{4,2} + 2y_{5,2} + o_1 + o_2 + 24$

by CGSS within the resource limits were filtered out, since the running time for checking an incomplete proof is inconclusive.

VeriPB successfully checked the proofs for 747 out of the 803 instances solved by CGSS (see Figure 2); 42 instances failed due to the memory limit and 14 instances failed due to the time limit. Checking the proof took about 3 times the solving time in the median for successfully checked instances. About 87% of the successfully checked instances were checked within 10 times the solving time.

Proof checking time compared to solver running time varies widely, but our experiments indicate that the performance of VeriPB is sufficient in most cases, and verification time scales linearly with the size of the proof for a majority of the instances. However, there is room to improve VeriPB, where focus so far has been on proof logging strength rather than performance. For the instances where checking is 100 times slower than solving, the main bottleneck is the proof generated by the SAT solver, which could be addressed by standard techniques for checking *DRAT* proofs, and checking logged solutions (when objective improving constraints (3a) are added) could also be implemented more efficiently.

Bugs Discovered by Proof Logging. Our work on implementing proof logging in CGSS led to the discovery of two bugs, which were also present in the solver RC2 on which CGSS is based, but have now been fixed in CGSS in commit `5526d04` and in RC2 in commit `d0447c3`. The bugs are due to a slightly different implementation of OLL compared to the description in Section 3.

First, when a counting variable $y_{K_{old},i}$ for a core K_{old} appears for the first time in a later core K_{new}, the next counting variable $y_{K_{old},i+1}$ is added to the reformulated objective with coefficient $w(K_{new}, O_{new})$ rather than $w(K_{old}, O_{old})$. The coefficient of $y_{K_{old},i+1}$ is then further increased when $y_{K_{old},i}$ is found in future cores. Second, rather than computing the upper bound UB from an actual solution, CGSS uses a weaker estimate UB_{est} obtained by summing the current lower bound and the

coefficients of all literals b where $coeff(O_{ref}, b) < w_{strat}$ (meaning that these literals were not set to 0 in the SAT solver call, and so could potentially be true in the solution).

The bugs we detected could lead to the solver producing an overly optimistic estimate $UB_{est} < UB$. The first way this can happen is when the contributions of counting variables $y_{K,k}$ in the reformulated objective are underestimated due to too small coefficients. The second bug is when the coefficient of $y_{K_{old},i+1}$ is first lowered below w_{strat} and then raised above this threshold again when $y_{K_{old},i}$ is found in a core. Then CGSS fails to assume $y_{K_{old},i+1} = 0$ in future solver calls. These bugs can result in erroneous hardening as detailed in the next example.

Example 1. Given a MaxSAT instance (F, O) with $F = \{(\bigvee_{i=1}^{5} b_i), (o_1 \vee o_2)\} \cup \{b_i \vee e_i \mid i = 1, \ldots, 5\}$ and $O = (\sum_{i=1}^{5} 10 \cdot b_i) + 11 \cdot e_1 + 14 \cdot e_2 + 11 \cdot e_3 + 3 \cdot e_4 + 2 \cdot e_5 + o_1 + o_2$, assume the stratification bound is $w_{strat} = 2$. Table 2 displays a possible CGSS run for this instance, except that for simplicity we assume one core extraction per iteration and no use of any other heuristics. The upper half of the table lists the variables set to 0 in solver calls, the extracted core, and the lower bound derived from it. The lower half of the table provides the reformulated objective. Even though the coefficient of $y_{K_1,3}$ is increased to 8 after the fourth core, this variable is not set to 0 in subsequent iterations, which allows the solver to finish the stratification level after extracting 6 cores with a solution that sets to true the variables $b_1, b_2, b_3, b_5, e_4, o_1, o_2, y_{K_2,2}$ and $y_{K_1,i}$ for $i = 1, \ldots, 4$, and all other variables to false. The cost of this solution is 45.

Now CGSS would incorrectly estimate $UB_{est} = LB + 4 = 28$, since $y_{K_1,3}$ and $y_{K_2,2}$ (abbreviated as $y_{1,3}$ and $y_{2,2}$ in the table) both have coefficient 1 in the current reformulated objective. This is lower than the cost 45 of the solution found (and even than the optimum 36), and erroneously allows hardening—which considers $y_{K_1,3}$ with the correct coefficient 8—to fix $y_{K_1,3} = 0$, even though b_1, b_2 and b_3 (and hence also $y_{K_1,3}$) are true in every minimal-cost solution.

In our computational experiments there were cases of faulty hardening, but all incorrectly fixed values happened to agree with some optimal solution and so we never observed incorrect results. Proof logging detected the problem, however, since the derivations of the buggy hardening steps failed during proof checking. Interestingly, what proof logging did *not* turn up was any examples of mistaken claims $O_{orig} \leq UB_{est} - 1$ when the cost of a found solution was estimated. The issue with mistaken estimates due to faulty stratification was instead discovered while analyzing and fixing the hardening bug. The moral of this is that even if all results are certified as correct, this does not certify that the code is free from bugs that have not yet manifested themselves. However, proof logging still guarantees that even if the solver would have undiscovered bugs, we can always trust computed results for which the accompanying proofs pass verification.

6 Concluding Remarks

In this work, we develop pseudo-Boolean proof logging techniques for core-guided MaxSAT solving and implement them in the solver CGSS [IBJ21] with support

for the full range of sophisticated reasoning techniques it uses. To the best of our knowledge, this is the first time a state-of-the-art MaxSAT solver has been enhanced to output machine-verifiable proofs of correctness. We have made a thorough evaluation on benchmarks from the MaxSAT Evaluation 2022 using the VeriPB proof checker [GN21, BGMN22], and find that proof logging overhead is perfectly manageable and that proof verification time, while leaving room for improvement, is definitely practically feasible. Our work also showcases the benefit of proof logging as a debugging tool—erroneous proofs produced by CGSS revealed two subtle bugs in the solver that previous extensive testing had failed to uncover.

Regarding proof verification time, further investigation is needed into the rare cases where verification is much slower (say, more than a factor 10) than solving. There are reasons to believe, though, that this is not a problem of MaxSAT proof logging per se, but rather is explained by features not yet added to VeriPB, which is a tool currently undergoing very active development. So far, the proof checker has been optimized for other types of reasoning than the clausal reverse unit propagation (RUP) steps that dominate SAT proofs. Also, VeriPB lacks the ability to trim proofs during checking as in [HHW13a]. Finally, introducing a binary proof format in addition to plain-text proofs would be another way to boost performance of proof checking. But these are matters of engineering rather than research, and can be taken care of once the proof logging technology as such has been developed and has proven its worth.

The focus of this work is on core-guided MaxSAT solving, but we would like to extend our techniques to solvers using linear SAT-UNSAT (LSU) solving (such as Pacose [PRB18]) and implicit hitting set (IHS) search (such as MaxHS [DB13a, DB13b]). Although there are certainly nontrivial technical challenges that will need to be overcome, we are optimistic that our work paves the way towards a unified proof logging system for the full range of modern MaxSAT solving approaches. Going beyond MaxSAT, it would also be interesting to extend VeriPB proof logging to pseudo-Boolean solvers using core-guided search [DGD+21] or IHS [SBJ21, SBJ22], and perhaps even to similar techniques in constraint programming [GBDS20] and answer set programming [AKMS12].

Acknowledgements

This work was partly carried out while some of the authors were visiting the Simons Institute for the Theory of Computing at UC Berkeley for the extended reunion of the program “Satisfiability: Theory, Practice, and Beyond” during the spring of 2023. We also benefited greatly from the Dagstuhl Seminar 22411 “Theory and Practice of SAT and Combinatorial Solving.” Additionally, we acknowledge several inspirational discussions on certifying solvers and proof logging with, among others, Ambros Gleixner, Stephan Gocht, and Ciaran McCreesh. The computational experiments were enabled by resources provided by LUNARC at Lund University.

Jeremias Berg was fully supported by the Academy of Finland under grant 342145. Bart Bogaerts and Dieter Vandesande were supported by Fonds Wetenschappelijk Onderzoek – Vlaanderen (project G070521N) and by the EU ICT-48

2020 project TAILOR (GA 952215). Jakob Nordström was supported by the Swedish Research Council grant 2016-00782 and the Independent Research Fund Denmark grant 9040-00389B. Andy Oertel was supported by the Wallenberg AI, Autonomous Systems and Software Program (WASP) funded by the Knut and Alice Wallenberg Foundation.

References

[ABGL12] Carlos Ansótegui, María Luisa Bonet, Joel Gabàs, and Jordi Levy. Improving SAT-based weighted MaxSAT solvers. In *Proceedings of the 18th International Conference on Principles and Practice of Constraint Programming (CP '12)*, volume 7514 of *Lecture Notes in Computer Science*, pages 86–101. Springer, October 2012.

[ABM+11] Eyad Alkassar, Sascha Böhme, Kurt Mehlhorn, Christine Rizkallah, and Pascal Schweitzer. An introduction to certifying algorithms. *it - Information Technology Methoden und innovative Anwendungen der Informatik und Informationstechnik*, 53(6):287–293, December 2011.

[ADR15] Mario Alviano, Carmine Dodaro, and Francesco Ricca. A MaxSAT algorithm using cardinality constraints of bounded size. In *Proceedings of the 24th International Joint Conference on Artificial Intelligence (IJCAI '15)*, pages 2677–2683. AAAI Press, 2015.

[AG17] Carlos Ansótegui and Joel Gabàs. WPM3: An (in)complete algorithm for weighted partial MaxSAT. *Artificial Intelligence*, 250:37–57, 2017.

[AGJ+18] Özgür Akgün, Ian P. Gent, Christopher Jefferson, Ian Miguel, and Peter Nightingale. Metamorphic testing of constraint solvers. In *Proceedings of the 24th International Conference on Principles and Practice of Constraint Programming (CP '18)*, volume 11008 of *Lecture Notes in Computer Science*, pages 727–736. Springer, August 2018.

[AKMS12] Benjamin Andres, Benjamin Kaufmann, Oliver Matheis, and Torsten Schaub. Unsatisfiability-based optimization in clasp. In *Technical Communications of the 28th International Conference on Logic Programming (ICLP '12)*, volume 17 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 211–221, September 2012.

[AW13] Tobias Achterberg and Roland Wunderling. Mixed integer programming: Analyzing 12 years of progress. In Michael Jünger and Gerhard Reinelt, editors, *Facets of Combinatorial Optimization*, pages 449–481. Springer, 2013.

[Bar95] Peter Barth. A Davis-Putnam based enumeration algorithm for linear pseudo-Boolean optimization. Technical Report MPI-I-95-2-003, Max-Planck-Institut für Informatik, January 1995.

[BB03] Olivier Bailleux and Yacine Boufkhad. Efficient CNF encoding of Boolean cardinality constraints. In *Proceedings of the 9th International Conference on Principles and Practice of Constraint Programming (CP '03)*, volume 2833 of *Lecture Notes in Computer Science*, pages 108–122. Springer, September 2003.

[BBN+23] Jeremias Berg, Bart Bogaerts, Jakob Nordström, Andy Oertel, and Dieter Vandesande. Experimental repository for "Certified core-guided MaxSAT solving". `https://doi.org/10.5281/zenodo.7709687`, May 2023.

[BGMN22] Bart Bogaerts, Stephan Gocht, Ciaran McCreesh, and Jakob Nordström. Certified symmetry and dominance breaking for combinatorial optimisation. In *Proceedings of the 36th AAAI Conference on Artificial Intelligence (AAAI '22)*, pages 3698–3707, February 2022.

[BHvMW21] Armin Biere, Marijn J. H. Heule, Hans van Maaren, and Toby Walsh, editors. *Handbook of Satisfiability*, volume 336 of *Frontiers in Artificial Intelligence and Applications*. IOS Press, 2nd edition, February 2021.

[Bie06] Armin Biere. Tracecheck. `http://fmv.jku.at/tracecheck/`, 2006.

[BJ17] Jeremias Berg and Matti Järvisalo. Weight-aware core extraction in SAT-based MaxSAT solving. In *Proceedings of the 23rd International Conference on Principles and Practice of Constraint Programming (CP '17)*, volume 10416 of *Lecture Notes in Computer Science*, pages 652–670. Springer, August 2017.

[BJM21] Fahiem Bacchus, Matti Järvisalo, and Ruben Martins. Maximum satisfiabiliy. In Armin Biere, Marijn J. H. Heule, Hans van Maaren, and Toby Walsh, editors, *Handbook of Satisfiability*, volume 336 of *Frontiers in Artificial Intelligence and Applications*, pages 929–991. IOS Press, 2nd edition, February 2021.

[BLM07] Maria Luisa Bonet, Jordi Levy, and Felip Manyà. Resolution for Max-SAT. *Artificial Intelligence*, 171(8-9):606–618, 2007.

[BMN22] Bart Bogaerts, Ciaran McCreesh, and Jakob Nordström. Solving with provably correct results: Beyond satisfiability, and towards constraint programming. Tutorial at the *28th International Conference on Principles and Practice of Constraint Programming*. Slides available at `http://www.jakobnordstrom.se/presentations/`, August 2022.

[BN21] Samuel R. Buss and Jakob Nordström. Proof complexity and SAT solving. In Biere et al. [BHvMW21], chapter 7, pages 233–350.

[BR07] Robert Bixby and Edward Rothberg. Progress in computational mixed integer programming—A look back from the other side of the tipping point. *Annals of Operations Research*, 149(1):37–41, February 2007.

[BS97] Roberto J. Bayardo Jr. and Robert Schrag. Using CSP look-back techniques to solve real-world SAT instances. In *Proceedings of the 14th National Conference on Artificial Intelligence (AAAI '97)*, pages 203–208, July 1997.

[CCT87] William Cook, Collette Rene Coullard, and György Turán. On the complexity of cutting-plane proofs. *Discrete Applied Mathematics*, 18(1):25–38, November 1987.

[CGSa] Certifying version of the CGSS core-guided MaxSAT solver with structure sharing. `https://gitlab.com/MIAOresearch/software/certified-cgss`.

[CGSb] CGSS, a core guided Max-SAT-algorithm using structure sharing technique for enhanced cardinality constraints, built on RC2 and PySAT. `https://bitbucket.org/coreo-group/cgss/`.

[CHH+17] Luís Cruz-Filipe, Marijn J. H. Heule, Warren A. Hunt Jr., Matt Kaufmann, and Peter Schneider-Kamp. Efficient certified RAT verification. In *Proceedings of the 26th International Conference on Automated Deduction (CADE-26)*, volume 10395 of *Lecture Notes in Computer Science*, pages 220–236. Springer, August 2017.

[CIP09] Chris Calabro, Russell Impagliazzo, and Ramamohan Paturi. The complexity of satisfiability of small depth circuits. In *Revised Selected Papers from the 4th International Workshop on Parameterized and Exact Computation (IWPEC '09)*, volume 5917 of *Lecture Notes in Computer Science*, pages 75–85. Springer, September 2009.

[CKSW13] William Cook, Thorsten Koch, Daniel E. Steffy, and Kati Wolter. A hybrid branch-and-bound approach for exact rational mixed-integer programming. *Mathematical Programming Computation*, 5(3):305–344, September 2013.

[CMS17] Luís Cruz-Filipe, João P. Marques-Silva, and Peter Schneider-Kamp. Efficient certified resolution proof checking. In *Proceedings of the 23rd International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS '17)*, volume 10205 of *Lecture Notes in Computer Science*, pages 118–135. Springer, April 2017.

[DB13a] Jessica Davies and Fahiem Bacchus. Exploiting the power of MIP solvers in MAXSAT. In *Proceedings of the 16th International Conference on Theory and Applications of Satisfiability Testing (SAT '13)*, volume 7962 of *Lecture Notes in Computer Science*, pages 166–181. Springer, July 2013.

[DB13b] Jessica Davies and Fahiem Bacchus. Postponing optimization to speed up MAXSAT solving. In *Proceedings of the 19th International Conference on Principles and Practice of Constraint Programming (CP '13)*, volume 8124 of *Lecture Notes in Computer Science*, pages 247–262. Springer, 2013.

[DGD+21] Jo Devriendt, Stephan Gocht, Emir Demirović, Jakob Nordström, and Peter Stuckey. Cutting to the core of pseudo-Boolean optimization: Combining core-guided search with cutting planes reasoning. In *Proceedings of the 35th AAAI Conference on Artificial Intelligence (AAAI '21)*, pages 3750–3758, February 2021.

[EG21] Leon Eifler and Ambros Gleixner. A computational status update for exact rational mixed integer programming. In *Proceedings of the 22nd International Conference on Integer Programming and Combinatorial Optimization (IPCO '21)*, volume 12707 of *Lecture Notes in Computer Science*, pages 163–177. Springer, May 2021.

[EGMN20] Jan Elffers, Stephan Gocht, Ciaran McCreesh, and Jakob Nordström. Justifying all differences using pseudo-Boolean reasoning. In *Proceedings of the 34th AAAI Conference on Artificial Intelligence (AAAI '20)*, pages 1486–1494, February 2020.

[ES06] Niklas Eén and Niklas Sörensson. Translating pseudo-Boolean constraints into SAT. *Journal on Satisfiability, Boolean Modeling and Computation*, 2(1-4):1–26, March 2006.

[FM06] Zhaohui Fu and Sharad Malik. On solving the partial MAX-SAT problem. In *Proceedings of the 9th International Conference on Theory and Applications of Satisfiability Testing (SAT '06)*, volume 4121 of *Lecture Notes in Computer Science*, pages 252–265. Springer, August 2006.

[FMSV20] Yuval Filmus, Meena Mahajan, Gaurav Sood, and Marc Vinyals. MaxSAT resolution and subcube sums. In *Proceedings of the 23rd International Conference on Theory and Applications of Satisfiability Testing (SAT '20)*, volume 12178 of *Lecture Notes in Computer Science*, pages 295–311. Springer, July 2020.

[GBDS20] Graeme Gange, Jeremias Berg, Emir Demirović, and Peter J. Stuckey. Core-guided and core-boosted search for constraint programming. In *Proceedings of the 17th International Conference on the Integration of Constraint Programming, Artificial Intelligence, and Operations Research (CPAIOR '20)*, volume 12296 of *Lecture Notes in Computer Science*, pages 205–221. Springer, September 2020.

[GMM+20] Stephan Gocht, Ross McBride, Ciaran McCreesh, Jakob Nordström, Patrick Prosser, and James Trimble. Certifying solvers for clique and maximum common (connected) subgraph problems. In *Proceedings of the 26th International Conference on Principles and Practice of Constraint Programming (CP '20)*, volume 12333 of *Lecture Notes in Computer Science*, pages 338–357. Springer, September 2020.

[GMN20] Stephan Gocht, Ciaran McCreesh, and Jakob Nordström. Subgraph isomorphism meets cutting planes: Solving with certified solutions. In *Proceedings of the 29th International Joint Conference on Artificial Intelligence (IJCAI '20)*, pages 1134–1140, July 2020.

[GMN22] Stephan Gocht, Ciaran McCreesh, and Jakob Nordström. An auditable constraint programming solver. In *Proceedings of the 28th International Conference on Principles and Practice of Constraint Programming (CP '22)*, volume 235 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 25:1–25:18, August 2022.

[GMNO22] Stephan Gocht, Ruben Martins, Jakob Nordström, and Andy Oertel. Certified CNF translations for pseudo-Boolean solving. In *Proceedings of the 25th International Conference on Theory and Applications of Satisfiability Testing (SAT '22)*, volume 236 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 16:1–16:25, August 2022.

[GN03] Evgueni Goldberg and Yakov Novikov. Verification of proofs of unsatisfiability for CNF formulas. In *Proceedings of the Conference on Design, Automation and Test in Europe (DATE '03)*, pages 886–891, March 2003.

[GN21] Stephan Gocht and Jakob Nordström. Certifying parity reasoning efficiently using pseudo-Boolean proofs. In *Proceedings of the 35th AAAI Conference on Artificial Intelligence (AAAI '21)*, pages 3768–3777, February 2021.

[GSD19] Xavier Gillard, Pierre Schaus, and Yves Deville. SolverCheck: Declarative testing of constraints. In *Proceedings of the 25th International Conference on Principles and Practice of Constraint Programming (CP '19)*, volume 11802 of *Lecture Notes in Computer Science*, pages 565–582. Springer, October 2019.

[HHW13a] Marijn J. H. Heule, Warren A. Hunt Jr., and Nathan Wetzler. Trimming while checking clausal proofs. In *Proceedings of the 13th International Conference on Formal Methods in Computer-Aided Design (FMCAD '13)*, pages 181–188, October 2013.

[HHW13b] Marijn J. H. Heule, Warren A. Hunt Jr., and Nathan Wetzler. Verifying refutations with extended resolution. In *Proceedings of the 24th International Conference on Automated Deduction (CADE-24)*, volume 7898 of *Lecture Notes in Computer Science*, pages 345–359. Springer, June 2013.

[IBJ21] Hannes Ihalainen, Jeremias Berg, and Matti Järvisalo. Refined core relaxation for core-guided MaxSAT solving. In *27th International Conference on Principles and Practice of Constraint Programming (CP '21)*, volume 210 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 28:1–28:19. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2021.

[IBJ22] Hannes Ihalainen, Jeremias Berg, and Matti Järvisalo. Clause redundancy and preprocessing in maximum satisfiability. In *Proceedings of the 11th International Joint Conference on Automated Reasoning (IJCAR '22)*, volume 13385 of *Lecture Notes in Computer Science*, pages 75–94. Springer, August 2022.

[IMM19] Alexey Ignatiev, António Morgado, and João P. Marques-Silva. RC2: an efficient MaxSAT solver. *Journal on Satisfiability, Boolean Modeling and Computation*, 11(1):53–64, September 2019.

[IP01] Russell Impagliazzo and Ramamohan Paturi. On the complexity of k-SAT. *Journal of Computer and System Sciences*, 62(2):367–375, March 2001. Preliminary version in *CCC '99*.

[KM21] Sonja Kraiczy and Ciaran McCreesh. Solving graph homomorphism and subgraph isomorphism problems faster through clique neighbourhood constraints. In *Proceedings of the 30th International Joint Conference on Artificial Intelligence (IJCAI '21)*, pages 1396–1402, August 2021.

[KP18] Michal Karpinski and Marek Piotrów. Competitive sorter-based encoding of PB-constraints into SAT. In *Proceedings of Pragmatics of SAT*, volume 59 of *EPiC Series in Computing*, pages 65–78. EasyChair, 2018.

[KP19] Michal Karpinski and Marek Piotrów. Encoding cardinality constraints using multiway merge selection networks. *Constraints*, 24(3-4):234–251, 2019.

[LBJ20] Marcus Leivo, Jeremias Berg, and Matti Järvisalo. Preprocessing in incomplete maxsat solving. In *Proceedings of the 24th European Conference on Artificial Intelligence (ECAI '20)*, volume 325 of *Frontiers in Artificial Intelligence and Applications*, pages 347–354. IOS Press, 2020.

[LM21] Chu Min Li and Felip Manyà. MaxSAT, hard and soft constraints. In *Handbook of Satisfiability*, volume 336 of *Frontiers in Artificial Intelligence and Applications*, pages 903–927. IOS Press, 2021.

[LNOR11] Javier Larrosa, Robert Nieuwenhuis, Albert Oliveras, and Enric Rodríguez-Carbonell. A framework for certified Boolean branch-and-bound optimization. *Journal of Automated Reasoning*, 46(1):81–102, 2011.

[LP10] Daniel Le Berre and Anne Parrain. The Sat4j library, release 2.2. *Journal on Satisfiability, Boolean Modeling and Computation*, 7:59–64, July 2010.

[LXC+22] Chu-Min Li, Zhenxing Xu, Jordi Coll, Felip Manyà, Djamal Habet, and Kun He. Boosting branch-and-bound MaxSAT solvers with clause learning. *AI Communications*, 35(2):131–151, 2022.

[MAGL11] João Marques-Silva, Josep Argelich, Ana Graça, and Inês Lynce. Boolean lexicographic optimization: algorithms & applications. *Annals of Mathematics and Artificial Intelligence*, 62(3-4):317–343, 2011.

[Max22] MaxSAT evaluation 2022. `https://maxsat-evaluations.github.io/2022`, August 2022.

[MDM14] António Morgado, Carmine Dodaro, and João P. Marques-Silva. Core-guided MaxSAT with soft cardinality constraints. In *Proceedings of the 20th International Conference on Principles and Practice of Constraint Programming (CP '14)*, volume 8656 of *Lecture Notes in Computer Science*, pages 564–573. Springer, September 2014.

[MIB+19] António Morgado, Alexey Ignatiev, María Luisa Bonet, João P. Marques-Silva, and Samuel R. Buss. DRMaxSAT with MaxHS: First contact. In *Proceedings of the 22nd International Conference on Theory and Applications of Satisfiability Testing (SAT '19)*, volume 11628 of *Lecture Notes in Computer Science*, pages 239–249. Springer, July 2019.

[MJML14] Ruben Martins, Saurabh Joshi, Vasco M. Manquinho, and Inês Lynce. Incremental cardinality constraints for MaxSAT. In *Proceedings of the 20th International Conference on Principles and Practice of Constraint Programming (CP '14)*, volume 8656 of *Lecture Notes in Computer Science*, pages 531–548. Springer, September 2014.

[MM11] António Morgado and João Marques-Silva. On validating Boolean optimizers. In *Proceedings of the 23rd IEEE International Conference on Tools with Artificial Intelligence, (ICTAI '11)*, pages 924–926, 2011.

[MMNS11] Ross M. McConnell, Kurt Mehlhorn, Stefan Näher, and Pascal Schweitzer. Certifying algorithms. *Computer Science Review*, 5(2):119–161, May 2011.

[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 Design Automation Conference (DAC '01)*, pages 530–535, June 2001.

[MS99] João P. Marques-Silva and Karem A. Sakallah. GRASP: A search algorithm for propositional satisfiability. *IEEE Transactions on Computers*, 48(5):506–521, May 1999. Preliminary version in *ICCAD '96*.

[NB14] Nina Narodytska and Fahiem Bacchus. Maximum satisfiability using core-guided maxsat resolution. In *Proceedings of the 28th AAAI Conference on Artificial Intelligence (AAAI '14)*, pages 2717–2723. AAAI Press, 2014.

[PCH20] Matthieu Py, Mohamed Sami Cherif, and Djamal Habet. Towards bridging the gap between SAT and Max-SAT refutations. In *Proceedings of the 32nd IEEE International Conference on Tools with Artificial Intelligence (ICTAI '20)*, pages 137–144, November 2020.

[PCH21] Matthieu Py, Mohamed Sami Cherif, and Djamal Habet. A proof builder for Max-SAT. In *Proceedings of the 24th International Conference on Theory and Applications of Satisfiability Testing (SAT '21)*, volume 12831 of *Lecture Notes in Computer Science*, pages 488–498. Springer, July 2021.

[PCH22] Matthieu Py, Mohamed Sami Cherif, and Djamal Habet. Proofs and certificates for Max-SAT. *Journal of Artificial Intelligence Research*, 75:1373–1400, December 2022.

[PRB18] Tobias Paxian, Sven Reimer, and Bernd Becker. Dynamic polynomial watchdog encoding for solving weighted MaxSAT. In *Proceedings of the 21st International Conference on Theory and Applications of Satisfiability Testing (SAT '18)*, volume 10929 of *Lecture Notes in Computer Science*, pages 37–53. Springer, July 2018.

[RvBW06] Francesca Rossi, Peter van Beek, and Toby Walsh, editors. *Handbook of Constraint Programming*, volume 2 of *Foundations of Artificial Intelligence*. Elsevier, 2006.

[SBJ21] Pavel Smirnov, Jeremias Berg, and Matti Järvisalo. Pseudo-Boolean optimization by implicit hitting sets. In *Proceedings of the 27th International Conference on Principles and Practice of Constraint Programming (CP '21)*, volume 210 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 51:1–51:20, October 2021.

[SBJ22] Pavel Smirnov, Jeremias Berg, and Matti Järvisalo. Improvements to the implicit hitting set approach to pseudo-Boolean optimization. In *Proceedings of the 25th International Conference on Theory and Applications of Satisfiability Testing (SAT '22)*, volume 236 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 13:1–13:18, August 2022.

[VDB22] Dieter Vandesande, Wolf De Wulf, and Bart Bogaerts. QMaxSATpb: A certified MaxSAT solver. In *Proceedings of the 16th International Conference on Logic Programming and Non-monotonic Reasoning (LPNMR '22)*, volume 13416 of *Lecture Notes in Computer Science*, pages 429–442. Springer, September 2022.

[Ver] VeriPB: Verifier for pseudo-Boolean proofs. `https://gitlab.com/MIAOresearch/software/VeriPB`.

Certifying Without Loss of Generality Reasoning in Solution-Improving Maximum Satisfiability

Paper III

Abstract

Proof logging has long been the established method to certify correctness of Boolean satisfiability (SAT) solvers, but has only recently been introduced for SAT-based optimization (MaxSAT). The focus of this paper is solution-improving search (SIS), in which a SAT solver is iteratively queried for increasingly better solutions until an optimal one is found. A challenging aspect of modern SIS solvers is that they make use of complex "without loss of generality" arguments that are quite involved to understand even at a human meta-level, let alone to express in a simple, machine-verifiable proof.

In this work, we develop pseudo-Boolean proof logging methods for solution-improving MaxSAT solving, and use them to produce a certifying version of the state-of-the-art solver Pacose with VeriPB proofs. Our experimental evaluation demonstrates that this approach works in practice. We hope that this is yet another step towards general adoption of proof logging in MaxSAT solving.

1 Introduction

Thanks to tremendous progress over the last decades on algorithms for combinatorial search and optimization, today *NP*-hard problems are routinely solved in many practical applications. Unfortunately, as these algorithms get more and more sophisticated, it also gets more and more challenging to avoid errors sneaking in during algorithm design and implementation. It is well-known that modern combinatorial solving algorithms in different paradigms can sometimes produce "solutions" that

Jeremias Berg, Bart Bogaerts, Jakob Nordström, Andy Oertel, Tobias Paxian, and Dieter Vandesand. "Certifying Without Loss of Generality Reasoning in Solution-Improving Maximum Satisfiability". In *Proceedings of the 30th International Conference on Principles and Practice of Constraint Programming (CP '24)*, volume 307 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 4:1–4:28, September 2024.

violate hard constraints, claim that suboptimal solutions are optimal, or declare that feasible problems lack solutions [BBN+23, BB09, BLB10, CKSW13, GSD19, JHB12].

Although there are many ways to address this problem, including software testing techniques such as fuzzing [BB09, PB23], and design of formally verified software [Fle20], the most promising approach appears to be the use of *certifying algorithms* [ABM+11, MMNS11] with so-called *proof logging*. What this means is the algorithm should not only produce an answer, but also a *proof* that this answer is correct. Such proofs should follow simple rules, as specified by a formal *proof system*, so that they can easily be verified by an independent *proof checker*. In addition to guaranteeing correctness, proof logging brings many other advantages: it enables advanced *testing* (since one can detect correct answers found for invalid reasons, and also test instances for which the answer is not known), detailed *debugging* (since invalid proof steps pinpoint where errors happened), *auditability* (since proofs can be stored and verified independently of which algorithm was used), and *performance analysis* (since proofs can be mined for insights on which reasoning steps were crucial for reaching the final conclusion).

Proof logging has been particularly successful in the domain of Boolean satisfiability (SAT) solving [BHvMW21], where a large variety of proof systems has seen the light of day [BCH21, Bie06, GN03, WHH14]. Using proof logging has long been mandatory in the main track of the SAT competitions, and it is hard to overestimate the impact this has had on improving overall correctness and reliability of SAT solvers. This has stimulated the spread of proof logging into other combinatorial solving paradigms, including SAT modulo theories (SMT) [SFBF21, BRK+22], automated planning [EH20, ERH17, ERH18, Rög17], and mixed integer linear programming [EG23, DEGH23].

Proof Logging for MaxSAT Solving In view of the above discussion, it is interesting to compare the developments in other combinatorial optimization paradigms to the state of affairs in maximum satisfiability (MaxSAT), the optimization version of the SAT problem. Without loss of generality, MaxSAT can be described as the problem of maximizing a linear objective O subject to satisfying a Boolean formula F in conjunctive normal form (CNF). Although MaxSAT is arguably the one optimization paradigm closest to SAT, and although several proof systems for formalizing MaxSAT reasoning have been proposed [BLM07, LNOR11, MM11, PCH20, PCH21, PCH22], for a long time there has been no practically feasible proof logging method for state-of-the-art MaxSAT solvers. This changed only recently when pseudo-Boolean proof logging using VeriPB [GN21, BGMN23] was proposed for MaxSAT [Van23, VDB22], a proposal that was followed by the successful design and implementation of VeriPB proof logging for modern core-guided MaxSAT solvers [BBN+23].

In this paper, we revisit proof logging work for *solution-improving search (SIS)* [Van23, VDB22], also referred to as *model-improving search* or *linear SAT-UNSAT (LSU) search*, and consider state-of-the-art solving techniques. In the SIS approach—which is much simpler to explain than, e.g., core-guided [FM06] or implicit hitting set [DB13] search—a SAT solver is repeatedly called on the formula F, each time with an added *solution-improving constraint* asking for increasingly better solutions with respect to the objective O, and the problem turns infeasible when the last

solution found was optimal. In the work by Vandesande et al. [Van23, VDB22], the main technical challenge was to certify correctness of the CNF encodings of these solution-improving constraints, which could then essentially be concatenated with the proof logging generated by the SAT solver (modulo some non-trivial engineering).

At first sight, it seems that implementing pseudo-Boolean proof logging in a state-of-the-art MaxSAT solver using solution-improving search would mostly be a matter of carefully transferring already developed techniques [Van23, VDB22], perhaps combining them with proof logging ideas developed for other CNF encodings [GMNO22]. After all, the distinguishing feature of a top-of-the-line SIS solver is the choice of CNF translation for reasoning about the objective function, such as, in the case of Pacose, the *polynomial watchdog (DPW)* encoding [BBR09]. Once proof logging for such a CNF encoding is in place, it seems reasonable to expect that the rest should be plain sailing.

It is all the more surprising, then, that it turns out nothing could be further from the truth. To minimize the time the MaxSAT solver spends on generating PW encodings, an essential step is to introduce completely unconstrained variables that can be used to perform different comparisons with a single CNF encoding; this is referred to as the *dynamic polynomial watchdog encoding (DPW)* [PRB18]. Loosely speaking, if we know that the best possible objective value lies in the range $[lo, hi]$, then instead of generating repeated encodings $O \geq V$ to probe different possible objective values V in this range, one can introduce free variables t_i encoding a tare sum T taking values between 0 and $hi - lo$ and try to maximize the value $T = T^*$ for which one single DPW-encoded constraint $O - T \geq lo$ holds. Once the maximum T^* has been found, it is clear that $O = lo + T^*$ is the best possible objective value, since without loss of generality T could be set to any value. But how can such a meta-argument be expressed in simple propositional logic reasoning?

In what follows, we provide a brief, if still high-level, discussion of some of the challenges that arise when trying to design simple proofs to certify such fairly complex "without loss of generality" arguments, and then outline how such challenges can be overcome.

Solution-Improving "Without Loss of Generality" Reasoning As already discussed above, the key aspect in which different solution-improving MaxSAT solvers differ is how they encode the solution-improving constraints. In order to compute the value of a linear expression L over 0–1 variables of interest, Pacose uses the polynomial watchdog encoding to describe a Boolean circuit BC with output variables z_k such that $z_k = 0$ implies $L \geq 1 + k \cdot 2^P$ (for some fixed integer P). If we chose L to be the objective function O that we are maximizing, this would allow to find the interval $\left[1 + k^* \cdot 2^P, (k^* + 1) \cdot 2^P\right]$ in which the optimal value lies by calling the SAT solver with the prechosen partial assignment $z_k = 0$ (referred to as an *assumption*) for increasing values of k until the solver returns that there is no satisfying assignment. To determine the exact location of the optimum in this interval, additional, completely unconstrained, variables t_i, called *tare variables*, are used to encode an integer $T = \sum_{i=0}^{P-1} 2^i t_i$ in the range $\left[0, 2^P - 1\right]$. The actual circuit in the encoding uses the linear form $L = O - T$, so that $z_k = 0$ means $O - T \geq 1 + k \cdot 2^P$. By making SAT solver calls with suitable assumptions on

the unconstrained t_i-variables, the optimal value of the objective function can be computed.

Given the CNF encoding of a circuit $\mathsf{BC}(O - T \geq 1 + k \cdot 2^P)$ evaluating the inequality $O - T \geq 1 + k \cdot 2^P$ as outlined above, the solution-improving search proceeds in two phases:

1. The *coarse convergence phase* identifies the largest k for which $z_k = 0$ is possible.

2. The *fine convergence phase* then maximizes the tare variable sum T.

Let us discuss this process in slightly more detail, and explain why it presents challenges from a proof logging point of view.

If during the coarse convergence phase a SAT solver call with assumption $z_k = 0$ returns a satisfying assignment α achieving objective value at least $1 + k \cdot 2^P$, the solver stores the information $z_k = 0$ (in the form of a unit clause $\overline{z}_k$), which enforces that any future solutions found have to be at least this good. The SAT solver is then called again with $z_{k'} = 0$ for some $k' > k$ to probe whether a solution exists with value at least $1 + k' \cdot 2^P$. Here it is relevant to note that fixing $z_k = 0$ could remove assignments corresponding to optimal solutions. For instance, if the optimal value is $V = V^* + 1 + k \cdot 2^P$, this value could be achieved by an assignment α' setting $T = T^* > V^* + 1$. For such an α' we would have $O - T = -T^* + V^* + 1 + k \cdot 2^P \leq k \cdot 2^P$, which would violate $z_k = 0$. However, since the tare variables are unconstrained, in this case there would also exist another assignment α'' achieving objective value $V^* + k \cdot 2^P$ for which $T = 0$, and so it is safe to require that solutions improving on α should satisfy $z_k = 0$.

In the fine convergence phase the z_k-variables are all fixed, and assumptions on the tare variables are made in the SAT solver calls to determine the exact value of the optimal solution. This again relies on reasoning without loss of generality, claiming that one can always choose $T \geq s$ for any value $0 \leq s < 2^P$. But now we are treading on dangerous ground: clearly, we cannot assume both $T = 0$ and $T \geq s > 0$ simultaneously! How can we convince ourselves, and more importantly, how can we convince a proof checker, that our derivations are consistent? At a meta-level, we can argue that since the tare variables are completely unconstrained in the original encoding, we should be able to fix them to any value we like at any given point in time. But how do we produce a simple, machine-verifiable proof that this is sound? And are we even sure this is sound?

Discussion of Our Contribution In this work, we show how pseudo-Boolean proof logging with VeriPB [GN21, BGMN23] can certify correctness of the complex CNF encodings used in state-of-the-art solution-improving MaxSAT solvers, as well as of the subtle without loss of generality reasoning applied on these encodings. To give a sense of how this can be done, we need to give a high-level description how VeriPB proofs work (referring the reader to later sections for the missing technical details).

A VeriPB proof maintains a set of *core constraints* $\mathcal{C}$, initialized to the formula F, together with a set of *derived constraints* $\mathcal{D}$ inferred by the solver. The proof semantics ensures that $\mathcal{C}$ and F have the same optimal value for O and that any solution to $\mathcal{C}$ can be extended to $\mathcal{D}$. A new constraint C can be derived "without

loss of generality" by the *redundance-based strengthening rule*, which requires the explicit specification of a substitution ω (mapping variables to truth values or literals) together with explicit proofs

$$\mathcal{C} \cup \mathcal{D} \cup \{\neg C\} \vdash (\mathcal{C} \cup \mathcal{D} \cup \{C\})\restriction_\omega \cup \{O\restriction_\omega \geq O\} \tag{1}$$

that all consequences on the right (with the substitution ω applied to the constraints) follow from previously derived constraints $\mathcal{C} \cup \mathcal{D}$ together with the negation $\neg C$ of the constraint to be inferred. This guarantees that if some assignment α satisfies everything so far but violates C, the "patched" assignment $\alpha \circ \omega$ satisfies also C and does not worsen the objective.

To make our informal discussion simple and concrete, suppose that we have a CNF encoding of a circuit $\mathsf{BC}(O - T \geq lo)$ evaluating $O - T \geq lo$, and that the solver has derived no constraints but only has the input formula F. If we want to fix $T = T^*$ using the redundance rule (1), we would have to find a substitution ω such that $F \cup \{\mathsf{BC}(O - T \geq lo)\} \cup \{T \neq T^*\}$ implies $\big(F \cup \{\mathsf{BC}(O - T \geq lo)\} \cup \{T = T^*\}\big)\restriction_\omega$. But it seems like this would force us to prove that if we take any assignment satisfying the Boolean circuit and modify the value of some of its inputs (the tares), the circuit would remain satisfied, and this is just not true. So although the redundance-based strengthening rule is very strong, it is not clear how it can be used to argue that the tare variables are unconstrained.

We get around this problem by first deriving a copy *shadow circuit* BC' of the original circuit, but substituting fixed values t_i^* for the tare variables, so that $\mathsf{BC}'(O - T^* \geq lo)$ evaluates $O - T^* \geq lo$. We then let ω be the substitution setting $t_i = t_i^*$ for all i and mapping all other variables x in BC to the corresponding shadow variables x' in BC', so that, effectively, the shadow circuit computes the substitution needed. This turns our application of the redundance rule (1) into

$$F \cup \{\mathsf{BC}(O - T \geq lo)\} \cup \{\mathsf{BC}'(O - T^* \geq lo)\} \cup \{T \neq T^*\} \tag{2a}$$

$$\vdash \big(F \cup \{\mathsf{BC}(O - T \geq lo)\} \cup \{\mathsf{BC}'(O - T^* \geq lo)\} \cup \{T = T^*\}\big)\restriction_\omega \cup \{O\restriction_\omega \geq O\} \tag{2b}$$

$$= F \cup \{\mathsf{BC}'(O - T^* \geq lo)\} \cup \{\mathsf{BC}'(O - T^* \geq lo)\} \cup \{T^* = T^*\} \cup \{O \geq O\} \tag{2c}$$

(where the final line (2c) is simply the result of applying the substitution ω to (2b)). If we study (2c) carefully, we see that all we need to prove about the circuit now is that the two copies of the shadow circuit in the consequences are implied by the same shadow circuit in the premises, and so (2c) follows trivially from the premises (2a).

This idea of using shadow circuits is crucial for certifying the correctness of assigning tare variables without loss of generality. However, we need to get rid of the completely unrealistic assumption that the solver would not have learned any constraints in $\mathcal{D}$. This is a problem in that the above argument fails when such learned constraints $D \in \mathcal{D}$ contain variables in the BC-circuit, since then there is no way to prove $D\restriction_\omega$ as required in (1).

Here a second idea discovered in recent VeriPB development turns out to be very helpful. Very briefly, it can be shown that if in the proof we enforce the requirement that all new constraints D derived by strengthening are immediately

moved to the core set $\mathcal{C}$, referred to as *strengthening-to-core*, then the redundance rule (1) can be simplified to

$$\mathcal{C} \cup \mathcal{D} \cup \{\neg C\} \vdash (\mathcal{C} \cup \{C\})\restriction_{\omega} \cup \{O\restriction_{\omega} \geq O\}, \tag{3}$$

omitting the proof obligations for the derived set $\mathcal{D}$. This means that we can ignore the problems arising from derived constraints when using shadow circuit reasoning.

We stress that this is only a brief and informal discussion that sweeps many technical challenges under the rug. Perhaps one of the most annoying such challenges is that the tare variables are sometimes fixed one at a time, and then a new shadow circuit is required for every new fixing. It would be desirable to find better ways of dealing with this problem.

We have implemented our methods in the state-of-the-art solution-improving MaxSAT solver Pacose [PRB18] to make it output VeriPB proofs, and have performed an extensive evaluation of how such proof logging works in practice. While there is certainly room for performance improvements in both proof generation and proof checking, the significance of our contribution is that we present practical methods to certify correctness for a solving paradigm that has previously been beyond the reach of proof logging. We hope that our work can serve as an impetus towards general adoption of proof logging for MaxSAT, and can stimulate further research on how to make these proof logging techniques more efficient.

As a final remark, we note that an interesting aspect of recent progress in proof logging is that it brings together all three software quality assurance methods discussed in the opening paragraphs above. While proof logging does seem like the most viable approach to certify correctness in combinatorial solving, extensive use of fuzzing techniques has been instrumental in our work to debug both proof logging routines and the VeriPB proof checker. This fuzzing, in turn, relies on the use of proof logging and on feedback from the proof checker. Finally, although we do not address this aspect in the current paper, formally verified proof checking backends as in [GMM+24, IOT+24] are crucially needed to ensure that the verdict of proof checkers for increasingly powerful proof logging systems can be trusted.

Outline of This Paper After reviewing some preliminaries in Section 2, we discuss the dynamic polynomial watchdog (DPW) encoding in Section 3. In Section 4 we describe how to design proof logging for solution-improving solvers using the DPW encoding, including a discussion of possible variations of our method (and of why simply using SAT proof logging for the final unsatisfiability call does not work). We report results from an empirical evaluation in Section 5 and end with some conclusions and a discussion of future research directions in Section 6.

2 Preliminaries

In this section, we review some pseudo-Boolean basics and then discuss MaxSAT in general and solution-improving search in particular, referring the reader to [BN21, LM21, BJM21] for more details.

Pseudo-Boolean Constraints and Proofs We write x to denote a $\{0,1\}$-valued Boolean variable, and write $\overline{x}$ as a shorthand for $1-x$, using ℓ to denote such *positive* and *negative literals*, respectively. A (linear) *pseudo-Boolean (PB) constraint C* is a 0–1 integer linear inequality $\sum_i w_i \ell_i \geq A$. Without loss of generality, we will often assume our constraints to be *normalized*, meaning that all literal are over distinct variables and the coefficients w_i and the *degree A* are non-negative. A *PB formula* is a conjunction of PB constraints.

A (disjunctive) *clause* is a PB constraint $\sum_i \ell_i \geq 1$ with all coefficients and degree equal to 1. We sometimes refer to constraints $\ell \geq 1$ with a single literal as *unit clauses* ℓ. We say that a formula is in *conjunctive normal form (CNF)* if it is a conjunction of clauses. A *(linear) pseudo-Boolean term* is a weighted sum $\sum_i w_i \ell_i$ of literals with integer coefficients. A *(partial) assignment* α is a (partial) function from variables to $\{0,1\}$; it is extended to literals by respecting the meaning of negation. We write $C\restriction_\alpha$ for the constraint obtained from C by substituting all assigned variables x by $\alpha(x)$ (and simplifying). A constraint C is *satisfied* under α if $\sum_{\alpha(\ell_i)=1} w_i \geq A$, and a formula F is satisfied if all its constraints are. We say that F *implies* C, denoted $F \models C$, if all assignments that satisfy F also satisfy C.

A *pseudo-Boolean optimization* (PBO) instance consists of a formula F and a linear term $O = \sum_i w_i \ell_i$ (called the *objective*). An assignment α to the variables in F and O that satisfies F is a *solution* to the instance, which is optimal if it *maximizes* the value $O\restriction_\alpha = \sum_i w_i \alpha(\ell_i)$.[1] For a PBO instance (F, O) the VeriPB proof system maintains a *proof configuration* of *core* and *derived constraints* $(\mathcal{C}, \mathcal{D})$, initialized to F and $\emptyset$, respectively. The VeriPB proofs we consider are in the so-called *strengthening-to-core* mode, which maintains the invariant that all constraints in the derived set $\mathcal{D}$ are implied by the core set $\mathcal{C}$. Constraints can be moved from $\mathcal{D}$ to $\mathcal{C}$ but not vice versa. New constraints can be derived from $\mathcal{C} \cup \mathcal{D}$ and added to $\mathcal{D}$ using the *cutting planes* proof system [CCT87] as follows:

Literal Axioms. For any literal ℓ_i, $\ell_i \geq 0$ is an axiom.

Linear Combination. Given two previously derived PB constraints C_1 and C_2, any positive integer linear combination of these constraints can be inferred.

Division. Given the normalized PB constraint $\sum_i w_i \ell_i \geq A$ and a positive integer c, the constraint $\sum_i \lceil w_i / c \rceil \ell_i \geq \lceil A/c \rceil$ can be inferred.

Some additional VeriPB proof rules extending cutting planes are as listed below—we refer to [BGMN23, GN21, HOGN24] for more details. For optimization problems we have rules for improvements of or rewriting of the objective function:

Objective Improvement. Given a total assignment α that satisfies $\mathcal{C} \cup \mathcal{D}$, one can add the constraint $O \geq 1 + O\restriction_\alpha$ to $\mathcal{C}$, which forces the search for strictly better solutions.

[1] Note that most of the PBO literature is formulated in terms of *minimization*, and this is also the perspective of VeriPB, but reasoning in terms of maximization is in line with the papers on solution-improving MaxSAT relevant for this work. We therefore adopt this perspective here, although the actual VeriPB proofs will argue in terms of minimizing the negation of the objective as described here.

Objective Reformulation. The current objective O can be replaced by a new objective O_{new} given explicit proofs from the core set $\mathcal{C}$ (using the VeriPB proof rules above) of the constraints $O - O_{\text{new}} \geq 0$ and $O_{\text{new}} - O \geq 0$ (i.e., a proof that $O = O_{\text{new}}$ holds).

Importantly, there are also rules for deriving non-implied constraints as long as the optimal value of the objective is preserved. VeriPB has a generalization of the RAT rule [JHB12] that makes use of *substitutions* ω, mapping variables to truth values or literals (where we extend the meaning of $C\!\upharpoonright_\omega$ to denote C with each x replaced by $\omega(x)$):

Redundance-Based Strengthening. The constraint C can be inferred and added to $\mathcal{C}$ by explicitly specifying a substitution ω and proofs $\mathcal{C} \cup \mathcal{D} \cup \{\neg C\} \vdash (\mathcal{C} \cup \{C\})\!\upharpoonright_\omega \cup \{O\!\upharpoonright_\omega \geq O\}$. This assumes strengthening-to-core mode—otherwise derivations for all constraints in $\mathcal{D}\!\upharpoonright_\omega$ are also needed (but then C can be placed in $\mathcal{D}$ instead of $\mathcal{C}$).

Intuitively, this rule shows that ω remaps any solution of $\mathcal{C}$ that does not satisfy C to a solution of $\mathcal{C}$ that satisfies also C without worsening the objective value. A typical use case of redundance-based strengthening is *reification*, which is the derivation of two pseudo-Boolean constraints that encode $\ell \Leftrightarrow D$ for some PB constraint D and for some fresh literal ℓ.

Finally, VeriPB has rules for deleting constraint in a way that guarantees that no spurious better-than-optimal solutions are introduced:

Deletion A constraint $D \in \mathcal{D}$ in the derived set can be deleted at any time. If strengthening-to-core mode is used, then deleting a constraint $C \in \mathcal{C}$ in the core set requires an explicit proof that C is implied by $\mathcal{C} \setminus \{C\}$. Otherwise, it is sufficient to show the weaker property that C can be derived from $\mathcal{C} \setminus \{C\}$ by redundance-based strengthening.

MaxSAT, Incremental SAT Solving, and Solution-Improving Search An instance of (weighted partial) Maximum Satisfiability (MaxSAT) consists of a CNF formula F and a pseudo-Boolean objective $O = \sum_i w_i \ell_i$ to be maximized under satisfying assignments to F, where we can assume without loss of generality that all literals in O are over distinct variables and that the constants are positive. Viewing MaxSAT in terms of an objective function and a CNF formula is equivalent to the more classical definition in terms of hard and soft clauses, in the sense that maximizing the objective corresponds to maximizing the total weight of satisfied soft clauses (see, e.g., [LBJ20] for more details).

The solution-improving search (SIS) algorithm we focus on in this work makes extensive use of incremental SAT solving with assumptions [ES03]. Invoking a SAT solver on a CNF formula F with a set of *assumptions* $\mathcal{A}$, i.e., a partial assignment, returns either 1. SAT and an extension of $\mathcal{A}$ that satisfies F or 2. UNSAT if no such assignment exists.

Given a MaxSAT instance (F, O), solution-improving search (SIS) computes an optimal solution by issuing a sequence of queries to a SAT solver asking for solutions of improving quality until an optimal one is found. More precisely, during search SIS maintains the best known solution α^*. In each iteration, the

algorithm queries a SAT solver on the working formula $F \wedge \mathsf{AsCNF}(O > O{\restriction_{\alpha^*}})$, where $\mathsf{AsCNF}(O > O{\restriction_{\alpha^*}})$ is a CNF formula that is satisfied by an assignment α if and only if it is a better solution than α^*, i.e., if $O{\restriction_\alpha} > O{\restriction_{\alpha^*}}$. If the SAT solver returns SAT, a better solution has been obtained and the working formula updated accordingly. Otherwise, if the SAT solver reports UNSAT, the best known solution α^* is determined to be optimal and the search is terminated.

The existing practical instantiations of SIS differ mainly in how the encoding of the formula $\mathsf{AsCNF}(O > O{\restriction_{\alpha^*}})$ is realized. Numerous CNF encodings of pseudo-Boolean constraints have been proposed for this task [ES06, JMM15, KP19, MPS14, Sin05]. For many instantiations of SIS the main challenge for proof logging is to certify the clauses added when encoding the objective constraint [VDB22, Van23], but as we will explain in the rest of this paper the so-called Dynamic Polynomial Watchdog encoding requires much more subtle arguments.

3 The Dynamic Polynomial Watchdog Encoding for SIS

The polynomial watchdog (PW) encoding [BBR09] is currently one of the best approaches for encoding pseudo-Boolean constraints in CNF, in terms of being compact while still propagating well. Using it for solution-improving search requires some non-trivial alternations, however, such as the addition of a dynamic constant. In this section we review this dynamic polynomial watchdog (DPW) encoding to the extent required for MaxSAT solution-improving search (SIS), referring the reader to [PRB18] for more details.

3.1 Initialization

Given a linear pseudo-Boolean term $L = \sum_i w_i \ell_i$, we define $w_{\max}$ to be the largest constant appearing in L. Additionally, we let $P := \lfloor \log_2(w_{\max}) \rfloor$ be one smaller than the number of bits in the binary representation of $w_{\max}$ and $W := \sum_i w_i$ be the maximum value for L. The polynomial watchdog encoding for L is a CNF formula $\mathsf{PW}(L)$ with $c := \lceil \frac{W}{2^P} \rceil$ output variables z_k for $k \in [0, c-1]$ enforcing the implications $\overline{z}_k \Rightarrow L \geq 1 + k \cdot 2^P$. In words, a satisfying assignment α of $\mathsf{PW}(L)$ that sets $\alpha(z_k) = 0$ will also satisfy $\sum_i w_i \alpha(\ell_i) \geq 1 + k \cdot 2^P$. We describe the formula $\mathsf{PW}(L)$ in more detail in Section 4.1.

Example 1. Consider a MaxSAT instance (F, O) and a working formula $F^w = F \wedge \mathsf{PW}(O)$. Assume we first invoke a SAT solver on F^w under the assumption $z_{k-1} = 0$ and then a second time under the assumption $z_k = 0$, and that the solver reports SAT for the first call and UNSAT for the second. At this point, we know that an optimal solution α^{opt} has value $O{\restriction_{\alpha^{\text{opt}}}}$ in the range $\left[1 + (k-1) \cdot 2^P, k \cdot 2^P\right]$.

The PW encoding was proposed as a way of enforcing a fixed bound B on the term L by considering a (static) constant $T = B - (1 + k \cdot 2^P)$, where k is the largest integer for which $B \geq 1 + k \cdot 2^P$, and encoding $\mathsf{PW}(L - T)$ [BBR09]. Then a solution that sets the kth output z_k of $\mathsf{PW}(L-T)$ to 0 will also satisfy $\sum_i w_i \alpha(\ell_i) - T \geq 1 + k \cdot 2^P$,

which is equivalent to $\sum_i w_i \alpha(\ell_i) \geq B$. The dynamic polynomial watchdog (DPW) encoding [PRB18] is an extension of the PW encoding that allows dynamically changing the value of T, and therefore also of B, so that the optimal value can be determined precisely with a single CNF encoding.

Consider a MaxSAT instance (F, O) and let $P = \lfloor \log_2(w_{\max}) \rfloor$ as described above. Instantiations of SIS with DPW introduce a "dynamic constant" in the form of a *tare* term $T := \sum_{i=0}^{P-1} 2^i \cdot t_i$, for fresh variables t_i not appearing anywhere else in the instance. The SAT solver is instantiated with the working formula $F \wedge \mathsf{PW}(O - T)$. Now we can use the output variables z_k to determine the optimal value within an additive constant 2^P, and then assign the tare T to values in $[0, 2^P - 1]$ to determine the precise value in that range. These are the *coarse convergence* and *fine convergence* phases mentioned in Section 1, which we describe in more detail next.

3.2 Coarse Convergence Phase

During the initial coarse convergence phase, only assumptions over the output variables z_k are made. Whenever a solution α is found, a call to the SAT solver is made with the assumption $z_k = 0$ where k is the largest natural number such that $O{\upharpoonright}_\alpha \geq 1 + (k-1) \cdot 2^P$. The coarse convergence phase ends when the solver reports UNSAT. The following observation summarizes the relevant conclusions of coarse convergence.

Observation 1. *Assume F is satisfiable and the SAT solver returns UNSAT under an assumption $z_{k^*} = 0$ in the coarse convergence phase. Then 1. there is a solution α^* to $F \wedge \mathsf{PW}(O - T)$ that assigns the tare variables so that $(O - T){\upharpoonright}_{\alpha^*} \geq 1 + (k^* - 1) \cdot 2^P$ holds, and 2. no solution β to F assigning also the tare variables can satisfy $(O - T){\upharpoonright}_\beta \geq 1 + k^* \cdot 2^P$.*

In words, coarse convergence provides bounds on the maximum value of $O - T$ obtainable by any solution of F. Importantly, as the tare term T is unconstrained by the formula F, its value can without loss of generality be assumed to be 0 at this stage, resulting in bounds on the objective value of optimal solutions as well. From now on, the algorithm commits to only searching for solutions that have $O - T$ in the specified interval, adding the unit clauses $\overline{z}_{k^*-1}$ and z_{k^*} to the working formula before proceeding to the fine convergence phase. In practice, whenever the SAT solver returns SAT after being called with assumption $\overline{z}_k$, the unit clause $\overline{z}_k$ is added immediately, allowing the SAT solver to simplify its clause database.

3.3 Fine Convergence Phase

During the fine convergence phase, assumptions for the tare variables are used to pinpoint the precise optimal value. Let k^* be the value for which the assumption $z_{k^*} = 0$ returned UNSAT in coarse convergence, and $o^* = O{\upharpoonright}_{\alpha^*}$ the objective value of the currently best known solution α^*. Then we define $s := o^* - (k^* - 1) \cdot 2^P$ to be the smallest value of the tare that would force an improved solution. The next call to the SAT solver assumes $t_i = 1$ for all tare variables for which the ith bit in the binary representation of s is 1. These assumptions enforce $T \geq s$, so any solution α

to the working formula (which now includes the unit clause $\overline{z}_{k^*-1} \geq 1$) that extends the assumptions will satisfy $O\restriction_\alpha \geq o^* + 1$.

The fine convergence phase continues in this manner until the SAT solver reports UNSAT, at which point an optimal solution has been found. As the value of s is monotonically increasing, we add unit clauses t_i to the working formula whenever we have deduced that the ith bit t_i in the tare T can safely be set to 1 in any solution (and hence in any future SAT call), which is the case when $s - 1 \geq 1 + \sum_{j=i}^{P-1} 2^i \cdot t_j$ holds. The fact that we have $s - 1$ rather than s in this last inequality is related to *stratification*, which we discuss next.

3.4 Stratification

Stratification is a technique for partitioning the indices of an objective $O = \sum_{i=1}^{m} w_i \ell_i$ into two sets $\{H, L\}$ in a way that allows computing the maximum values first of $O_H = \sum_{i \in H} w_i \ell_i$ and then of $O_L = \sum_{i \in L} w_i \ell_i$, and finally combining them to get the maximum value of O.

Specifically, stratification is applied when $\gcd\{w_i \mid i \in H\} \geq \sum_{i \in L} w_i$, i.e., when the greatest common divisor of the coefficients in O_H is at least the sum of all coefficients in O_L. SIS with the DPW encoding and stratification will first run coarse and fine convergence only on O_H as described above. At the end of the fine convergence, the SAT solver returns UNSAT after being invoked with assumptions that enforce $T_H \geq s$ for the tare term T_H added to the DPW encoding of O_H and some constant s. At this stage, the value of T_H will be fixed to $s-1$ with unit clauses, effectively fixing O_H to its maximum value. This fixing of O_H is consistent with the unit clauses learned in the previous section. After this O_L is optimized via coarse and fine convergence under the fixed value of O_H. The solution obtained at the end of the final fine convergence phase will be optimal with respect to the original instance. For more details on stratification, we refer the reader to [ALM09, PRB21].

Example 2. Consider the objective $O = 10x_1 + 5x_2 + 5x_3 + 3x_4 + 2x_5$ and the partition $H = \{1, 2, 3\}$ and $L = \{4, 5\}$. Since $\gcd\{10, 5, 5\} = 5 \geq 3 + 2$, changes of the objective restricted to $\{x_1, x_2, x_3\}$ will dominate any contributions from $3x_4 + 2x_5$. If a solution α with $O_H\restriction_\alpha = 15$ is found, we can without loss of generality assume $O_H \geq 15$, since for any solution β with $O_H\restriction_\beta < 15$ we have $O\restriction_\beta \leq O\restriction_\alpha$. Notice that maximizing first O_H and then O_L can remove some optimal solutions from the search space, but never all of them.

4 Certifying Solution-Improving MaxSAT with the DPW Encoding

We are now ready to describe how to do proof logging for solution-improving MaxSAT with the dynamic polynomial watchdog encoding. In addition to certifying the correctness of CNF encodings, as done in previous work on proof logging SIS for MaxSAT [VDB22, Van23], we need to certify the without loss of generality reasoning discussed in Section 3. This turns out to require quite intricate proof logging methods.

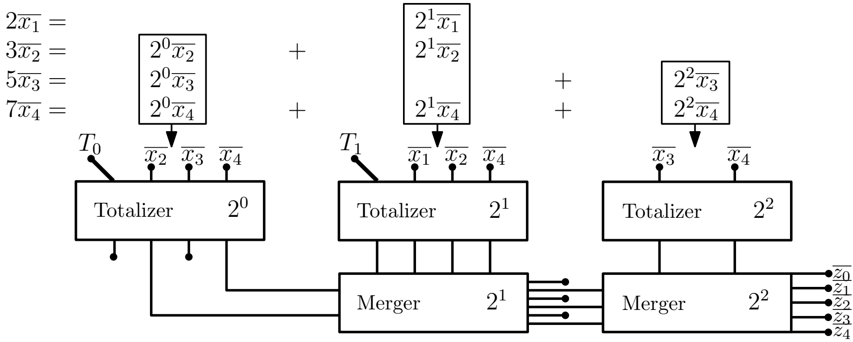


Figure 1: *Illustration of the polynomial watchdog encoding.*

We start with a brief discussion how to certify the DPW encoding. We then turn to proof logging for the without loss of generality reasoning during the coarse and fine convergence phases. Afterwards, we deal with proof logging for stratification. We defer a discussion of minor additional heuristics used in state-of-the-art solvers to Appendix B. We note that for all clauses learned by the SAT solver we can use standard VeriPB proof logging, and since all such learned clauses are logically implied by the working formula it is safe to add them to the derived set $\mathcal{D}$. This means that we can ignore all constraints added to the database by the SAT solver when we perform redundance-based strengthening steps.

4.1 Proof Logging for Clauses of the DPW Encoding

Figure 1 depicts the structure of the DPW encoding of the term $2x_1+3x_2+5x_3+7x_4$. For a term L in which the largest coefficient has P bits, the encoding introduces P totalizers [BB03] (which are circuits that sort their inputs), and $P-1$ mergers. The ith totalizer takes as input all variables in L for which the corresponding coefficient has its ith bit equal to 1.

Proof logging for the DPW encoding boils down to taking care of the totalizer encodings as described in [VDB22]. At a high level, the proof for $\mathsf{PW}(O-T)$ derives a number of constraints encoding implications $y \Rightarrow C_y$ and $y \Leftarrow C_y$, where y are variables in the auxiliary variable set Y and C_y are suitably chosen PB constraints over the variables in $O-T$. A concrete example is the output variable z_k for which the constraint C_{z_k} is chosen as $O - T \leq k \cdot 2^P$. From these pseudo-Boolean definitions all clauses in the CNF encoding added to the solver database can be derived with explicit VeriPB derivations. A technical point that is crucial for the proof logging is that in this way we only need to add the PB definitions of new variables to the core set $\mathcal{C}$. The clauses actually used for the SAT solver calls are implied from these definitions, and can therefore be placed in the derived set $\mathcal{D}$.

4.2 Proofs Without Loss of Generality Using Shadow Circuits

The MaxSAT solving algorithm uses without loss of generality (wlog) reasoning when 1. introducing fresh variables for encoding $\mathsf{PW}(O-T)$; 2. adding unit

clauses $\overline{z}_k$ during coarse convergence; 3. learning unit clause over the tare variables t_i during fine convergence; and 4. concluding that the optimal value has been found.

To see why unit clauses $\overline{z}_k \geq 1$ require wlog reasoning, suppose in the coarse convergence phase that the SAT solver returns a solution α when invoked with the assumption $z_k = 0$, indicating that $(O - T){\upharpoonright}_\alpha \geq 1 + k \cdot 2^P$. The constraint $\overline{z}_k \geq 1$ is *not* entailed by the solution-improving constraint $O \geq O{\upharpoonright}_\alpha$, since some other (possibly optimal) solution β might have $O{\upharpoonright}_\beta \geq O{\upharpoonright}_\alpha$ but assign the tare variables so that $(O - T){\upharpoonright}_\beta < 1 + k \cdot 2^P \leq (O - T){\upharpoonright}_\alpha$ holds. However, since the tare variables are not constrained by the original formula F, any solution to F could be extended to any fixed value for the tare T. Hence, in particular, we can assume without loss of generality that $T = 0$, which in turn implies that $\overline{z}_k \geq 1$.

The fine convergence phase makes use of the fact that the DPW encoding does not constrain T, which takes values in the range $\left[0, 2^P - 1\right]$. The unit clauses $t_i \geq 1$ learned are not entailed, but can be deduced since the tare variables are unconstrained in the DPW encoding. This requires a VeriPB proof that wlog $T \geq s - 1$. When the SAT solver reports UNSAT during fine convergence, it does so under the assumption that a specific set of tare variables take value 1. If this yields UNSAT, then we can conclude that the current solution is optimal (since we can wlog assume T to be equal to the value that led to UNSAT).

It is worth noticing that the without loss of generality arguments above are quite intricate even at a human meta-level. The coarse convergence phase repeatedly claims to be able to assume $T = 0$, after which the fine convergence phase picks an increasing sequence $0 < s_1 < s_2 < \ldots$ and assumes $T \geq s_i - 1$ wlog. Finally, a specific value $T = s_{i^*}$ is used to argue about optimality. The meta-level argument for why this works is that no conclusions are drawn from the assumptions made during coarse and fine convergence that invalidate subsequent assumptions. The challenge is how to convince a mechanical proof checker of this.

Consider first proof logging for the coarse convergence phase, and suppose the solver returns SAT when invoked with assumption $\overline{z}_k$. The only rule that would allow us to derive $\overline{z}_k \geq 1$ without loss of generality (from the argument that we can set $T = 0$ wlog) is *redundance-based strengthening*, which requires specification of a witness substitution ω that can be used to "patch" any assignment α in which $\overline{z}_k \geq 1$ is violated. More formally, our witness should guarantee that $\mathcal{C} \cup \mathcal{D} \cup \{\neg(\overline{z}_k \geq 1)\} \models \left(\mathcal{C} \cup \{\overline{z}_k \geq 1\}\right){\upharpoonright}_\omega \cup \{O \leq O{\upharpoonright}_\omega\}$. A natural approach would be to choose a witness ω that maps 1. z_k to 0, 2. all original variables to themselves, and 3. T to 0. Such a witness would make $(\overline{z}_k \geq 1){\upharpoonright}_\omega$ trivially true and would incur no proof obligations for the formula F or the objective O. However, setting $T = 0$ will not work for the constraints $C \in \mathcal{C}$ defining variables in the DPW encoding. If we fix $T = 0$, then we also need to update all auxiliary variables Y in the circuit evaluating $\mathsf{PW}(O - T)$. But how this should be done depends on which assignment α we need to patch, and the redundance rule has no mechanism for defining "conditional witnesses" $\omega = \omega(\alpha)$.

To determine how the witness should assign the auxiliary variables in $\mathsf{PW}(O-T)$, we devise a new proof logging technique that we call *shadow circuits*. Corresponding to each auxiliary variable y defined as the reification of a PB constraint C_y in the original circuit, a *shadow circuit for a fixed value v* has a fresh variable $y^{T=v}$ defined

by $y^{T=v} \Leftrightarrow C_y\restriction_{T\mapsto v}$. In words, the defining constraints of $y^{T=v}$ and y are the same except that we fix the tare variables t_i so that $T = v$. The definitions of such shadow circuits are stored in the core set $\mathcal{C}$ since they are derived using the redundance rule. Note that the shadow circuit only "copies" the pseudo-Boolean definitions of the variables and not their clausal encodings.

Shadow circuits provide us with a mechanism to compute witnesses for the redundance rule that allow us to assume the value of T and certify the without loss of generality reasoning. During coarse convergence, each addition of a constraint $\overline{z}_i \geq 1$ is logged with a witness that maps all tare variables t_i to 0 and other auxiliary variables y in $\mathsf{PW}(O-T)$ to their counterparts $y^{T=0}$ in the shadow circuit for $T = 0$. During fine convergence, the constraints $T \geq s-1$ are derived using shadow circuits for $s-1$, which allows adding unit constraints over individual tare variables to the proof. Finally, for proving optimality a shadow circuit for the final value s^* for which the SAT solver returned UNSAT will be used to derive contradiction.

The next proposition gives a more formal summary of the wlog proof logging performed during the coarse convergence phase. The proof for this proposition, together with precise descriptions of the other wlog proof logging steps, are given in Appendix A.

Proposition 2. *Suppose the* VeriPB *proof log contains derivations of reification constraints* $\overline{z}_k \Leftrightarrow O - T \geq 1 + k \cdot 2^P$ *and a shadow circuit for* $T = 0$ *as well as the constraint* $O \geq 1 + k \cdot 2^P$. *Then the constraint* $\overline{z}_k \geq 1$ *can be derived using redundance-based strengthening with witness* $\omega = \{t_i \mapsto 0 \mid 0 \leq i \leq P-1\} \cup \{y \mapsto y^{T=0} \mid y \in Y\}$.

The constraint $O \geq 1 + k \cdot 2^P$ in Proposition 2 can be obtained by weakening the solution-improving constraint $O \geq O\restriction_\alpha + 1$ for the previously found solution α. If stratification is used, deriving $O_H \geq 1 + k \cdot 2^P$ requires more work (see Section 3.4 for details).

Our technique with shadow circuits and repeated without loss of generality arguments selecting (different) values for the same variables in T heavily relies on that VeriPB proofs in the *strengthening-to-core* mode maintain the guarantee that all constraints in the derived set $\mathcal{D}$ are entailed by the core set $\mathcal{C}$. In particular, what this means is that whenever we want to apply redundance-based strengthening, fixing tare variables and using the corresponding shadow circuit, we do not need to worry about reproving any clauses learned by the SAT solver under the witness ω. It turns out that for all non-trivial proof obligations, the solution-improving constraint $O \geq O\restriction_\alpha$ for the latest solution α obtained is helpful. This also makes it easier to see why the entire pipeline is consistent. During coarse convergence, we never derive $T = 0$, but instead derive $z_k = 0$ for certain values of k using the fact that we could set $T = 0$ wlog. This constraint $z_k = 0$ will be used by the solver for deriving several consequences. Later, when we make the wlog argument that $T \geq s-1$ for some value s, this incurs the obligation to reprove that $z_k = 0$ holds! That is, the proof checker realizes that $z_k = 0$ was also derived wlog, and we need to prove that this is still consistent with the current wlog assumption to justify that we can "change our mind" about the value of T.

The use of *strengthening-to-core* requires some extra care when dealing with constraint deletions. SAT solvers use heuristics to aggressively erase clauses that

are believed to no longer be useful, and this is crucial for performance. Also, clauses in the input are removed whenever some literal in the clause is deduced to be true. In strengthening-to-core mode, we can still do unrestricted deletions of constraints in the derived set $\mathcal{D}$, but a core constraint $C \in \mathcal{C}$ can only be erased if the implication $\mathcal{C} \setminus \{C\} \models C$ can be shown to hold. For this reason we did not implement deletion from the core set in our proof logging routines.

4.3 Stratification

For proof logging of stratification steps as in Section 3.4, we need to be able to convert known facts about the whole objective O to statements about the split objectives O_H and O_L. To certify a unit constraint added during coarse convergence or to derive the constraints $T \geq s - 1$ during fine convergence when maximizing O_H, we need to derive $O_H \geq O_H{\upharpoonright_\alpha}$ from $O \geq O{\upharpoonright_\alpha} + 1$. We do this by weakening away all terms in O_L—meaning that for every term $w_i \ell_i$ in O_L we add $w_i \overline{\ell_i} \geq 0$ to cancel the term—to get $O_H \geq O{\upharpoonright_\alpha} + 1 - g$, where g is the greatest common divisor of the coefficients in O_H. This clearly also entails $O_H \geq O_H{\upharpoonright_\alpha} - g + 1$. Dividing by g and rounding up yields $\frac{1}{g} O_H \geq \frac{O_H{\upharpoonright_\alpha}}{g} - 1 + 1$, and multiplying this again by g yields $O_H \geq O_H{\upharpoonright_\alpha}$.

By applying this reasoning, we can derive the constraint $O_H \geq o_H^*$ right after finding the optimal value o_H^* for O_H. Moreover, after introducing a shadow circuit for $T = s$, we can derive (local) optimality in the form of the constraint $O_H \leq o_H^*$. Hence, we can reformulate the objective by replacing O_H with the constant o_H^*, from which we can now derive the constraint $O_L + o_H^* \geq O{\upharpoonright_\alpha} + 1$. Observe that this constraint coincides with the solution-improving constraint for O_L. Once the constraints $O_L \geq o_L^*$ and $O_L \leq o_L^*$ have been derived in a similar way, the objective will be rewritten to a constant, for which proving optimality boils down to logging a solution that has objective value $o^* = o_H^* + o_L^*$.

4.4 Limiting the Use of Shadow Circuits

Our proof logging method makes repeated use of shadow circuits, which are copies of the original circuit, and repeatedly deriving all constraints defining such circuits could potentially incur serious overhead for proof generation in the solver. Let us discuss ways of limiting or completely eliminating the use of shadow circuits and the downside of such approaches.

First, the shadow circuits are introduced each time the solver deduces a unit clause over an output variable z_k or tare variable t_i. Instead of learning these unit clauses, we could do all subsequent solver calls with those literals as assumptions. At the very end of the fine convergence phase, we could then introduce a single shadow circuit to prove optimality (or, in case of stratification, two shadow circuits: one to prove optimality and one to fix the value of the tare variables). The disadvantage is that when variables used as assumptions, the solver cannot use them to simplify its clause database; so while this would have a positive effect on the time required to do the actual proof logging, it could have negative effects on solving time. Appendix C.2 reports on an experimental evaluation of this approach.

Second, there is a way to completely eliminate shadow circuits. By the end of the execution, the solver knows which value $T = s$ resulted in the final *UNSAT* call in the fine convergence. What we could do at this point is insert at the *beginning* of the proof constraints saying that $T = s$ holds (which at this point can easily be derived by redundance-based strengthening). The rest of the proof will then be checked for a fixed value of T that happens to be the value needed at the end. There are two important reasons why we prefer the shadow circuit approach. The first reason is that it is not clear if and how this would work together with stratification, where after a stratification level we want to fix $T = s - 1$. The second reason is that fixing T in advance adds substantial new information that the solver did not have available when constructing the proof. This means that we would not be verifying that the reasoning the solver actually performed was correct, but only that its reasoning checks out given advance information about the optimal solution. While this could still be used to certify the correctness of the final answer, it would not provide any guarantees about the process leading there. It has been shown repeatedly that proof logging can catch subtle bugs in solvers that only report correct results but for the wrong reasons [EG23, GMM+20, KM21, BBN+23], but in order for this to be possible the correctness of solver-generated proofs should only depend on what the solver actually knows when the proof is being produced.

4.5 Discussion of an Even Simpler Approach and Why It Does Not Work

The proof logging techniques in this paper certify every single reasoning step in the solver. An alternative, and seemingly much simpler, way to get proofs of correctness for *any* MaxSAT solver would be to (i) compute an optimal solution by running the MaxSAT solver without proof logging, (ii) check that this solution is feasible, (iii) encode a solution-improving constraint into CNF, and (iv) call a SAT solver to generate a proof of unsatisfiability (and hence of optimality of the solution) with standard SAT proof logging. However, there are several serious issues with this approach that we would like to point out.

First, proofs of correctness are needed for the CNF encodings used in step (iii), and such proofs cannot be done with SAT proof logging since it cannot reason about values of objective functions. Second, it is not possible to just repeat the "final UNSAT call" of the MaxSAT solver in step (iv). Even if the same SAT solver is used, in the original UNSAT call this solver had access to all constraints learned in previous calls, and there is no guarantee that the solver will learn these constraints again, or other equally good constraints, when it is now run in a different way and with a different input. It is therefore impossible to know for sure whether the final SAT solver invocation with the solution-improving constraint would be faster or, more likely, slower, than the original solving process, and by how much. This defeats the whole idea of generating proofs with a small and predictable overhead, since there would be no way of knowing in advance whether "proof logging" for a previously claimed result would succeed or not. Moreover, when a solution-improving MaxSAT solver makes use of stratification (as discussed in Section 3.4), then optimality is not derived by a single UNSAT call but by

a combination of UNSAT calls at different levels. It is hard to see how such a combination of calls could be replicated with the simple approach described above.

Third, an increasingly popular usage scenario for MaxSAT solvers is so-called anytime solving, where the solver can be terminated at any point and then returns the best upper and lower bounds on the objective computed so far. Proofs constructed as described in this paper (as well as in other MaxSAT papers using VeriPB proof logging) will at all times contain formal proofs of everything the solver knows about upper and lower bounds on the objective. Whenever the solver is terminated, it can therefore just end the generated proof at that point by printing a concluding line stating what upper and lower bounds have been proven. This functionality would be lost in the alternative approach.

Finally, even if this approach could be made to work efficiently—which, as explained above, is not really the case, for several reasons—we would have the same problem as in Section 4.4 that we would only certify the final result and not the solver reasoning process.

5 Experimental Evaluation

To evaluate our proof logging approach in practice, we implemented it in the state-of-the-art solution-improving MaxSAT solver Pacose [PRB18]. The source code for all software tools used, as well as all experimental data, are available in [BBN+24]. During development, we extensively checked the correctness of our implementation with a fuzzer [PB23] and minimized failed instances with a delta debugger. This process accelerated the development, as we did not need to create instances for special cases, and helped us fix unexpected and sporadic bugs. The proofs emitted by Pacose were verified by the pseudo-Boolean proof checker VeriPB [Ver], and our fuzzing also helped to debug the proof checker.

The experiments were performed on identical machines with an 11th Gen Intel(R) Core(TM) i5-1145G7 @ 2.60 GHz CPU and 16 GB of memory. Each benchmark ran exclusively on a machine and the memory limit was set to 14 GB. The time limits were set to 3 600 seconds for solving a MaxSAT instance with Pacose and to 36 000 seconds for checking the proof with VeriPB. As our benchmark set we used the 558 weighted and 572 unweighted MaxSAT instances from the MaxSAT Evaluation 2023 [Max23].

Our implementation supports all techniques Pacose employed in the MaxSAT Evaluation 2023. This means that in addition to the dynamic polynomial watchdog encoding we also implemented proof logging for the binary adder encoding [War98] following the approach in [GMNO22, Van23] as well as support for stratification as described in Section 3.4 and for the preprocessing techniques in TrimMaxSAT [PRB21]. Appendix B discusses TrimMaxSAT in detail and Appendix C contains detailed experimental results for the default setup in which Pacose employs heuristics to choose between different encodings. In this section, we focus on the main novelty of this paper, namely proof logging for SIS with the DPW encoding.

To show the viability of enabling proof logging while solving, we analyse the overhead of generating proofs. In Figure 2 we compare the running time of Pacose

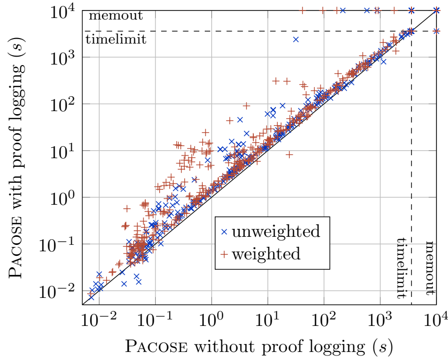


Figure 2: *Proof logging overhead for PACOSE using the DPW encoding.*

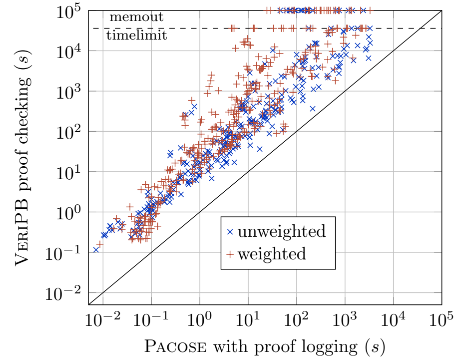


Figure 3: *PACOSE vs. VERIPB running time using DPW encoding.*

with and without proof logging. With proof logging enabled 674 instances were solved within the resource limits, which is 11 fewer instances than without proof logging. Out of the 11 instances that were not solved with proof logging enabled, 9 instances failed due to the memory limit and 2 instances due to the time limit. For the solved instances, PACOSE with proof logging was on average 1.93× slower than without proof logging. About 90% of the solved instances were solved at most 5.26× more slowly with proof logging enabled. This overhead for solving is to some extent caused by our shadow circuits approach. While we demonstrate that shadow circuits can be used to justify the without loss of generality reasoning in PACOSE, it remains to investigate whether there is a better approach. It is important to note, though, that the average overhead of 1.93× is heavily biased by small instances: the cumulative solving time of all 674 instances, with proof logging is only 1.32× the cumulative solving time without proof logging. This suggests that proof logging overhead decreases for harder instances.

For proof logging to be maximally useful in practice, it is also desirable that it should be possible to check generated proofs within a time limit that is some small constant factor of the solving time for the instance. To evaluate the efficiency of proof checking, we compared the running time of PACOSE with proof logging enabled with the running time of VERIPB, with results plotted in Figure 3. Out of the 674 instances solved by PACOSE with proof logging, 592 were successfully checked by VERIPB, but 53 instances failed due to the memory limit and 29 instances due to the time limit. On average, checking the proof with VERIPB was 22.5× slower than solving and generating the proof with PACOSE. 90% of the proofs were checked within 100× the running time of PACOSE. These results for checking are in line with what has been reported in other works on proof logging for MaxSAT [BBN+23, Van23]. While there is certainly room for further improvements, this shows that proof logging and checking is viable. It should also be emphasized that the only sources of problems for VERIPB were the time and memory limits—other than that all proofs were successfully checked.

6 Conclusion

In this paper, we demonstrate how to design proof logging for solution-improving MaxSAT solving using the dynamic polynomial watchdog encoding. This turns out to be surprisingly challenging, mainly due to the heavy use of reasoning without loss of generality. To understand the correctness of this reasoning at a human level is one thing, but convincing a proof checker by producing machine-verifiable proofs is quite another. What we show is that by combining the redundance-based strengthening rule and the strengthening-to-core mode in VeriPB, together with a technique we call shadow circuits for having more expressive witnessing capabilities, we are able to devise efficient pseudo-Boolean proof logging techniques.

We have implemented our approach in the state-of-the-art MaxSAT solver Pacose. Our experimental evaluation shows that while enabling proof logging is feasible, it does incur a non-negligible overhead in solving time. Moreover, the time needed to check the generated proofs is several times larger than the time needed to generate them, suggesting that more efficient algorithms and more optimized engineering are needed in VeriPB. This is not so surprising, since the focus of VeriPB development so far has been on providing support for certifying algorithms in combinatorial optimization paradigms previously beyond the reach of proof logging, rather than on optimizing the proof checker code base.

The addition of Pacose to the collection of certifying MaxSAT solvers using VeriPB proofs provides further support to the hypothesis that pseudo-Boolean proof logging hits a sweet spot for MaxSAT solving, being rich enough to support a wide variety of solving algorithms and complex reasoning tricks, but still being simple enough to support even formally verified proof checking as in [BMM+23, GMM+24, IOT+24].

We believe that in the longer term VeriPB can have a strong positive impact on the reliability and robustness of MaxSAT solvers. In the other direction, MaxSAT solving is likely to provide excellent benchmarks and performance challenges to further improve pseudo-Boolean proof logging and checking. Our suggestion for speeding up these developments is to introduce a certifying track in the yearly MaxSAT Evaluation [Max].

Acknowledgements

We want to thank Florian Pollitt and Mathias Fleury for their assistance with the CaDiCaL proof tracer and for fuzzing VeriPB within CaDiCaL. Their contributions were very helpful to further improve the robustness of the VeriPB toolchain. We also wish to acknowledge useful discussions with participants of the Dagstuhl workshop 23261 *SAT Encodings and Beyond*. The computational experiments were enabled by resources provided by LUNARC at Lund University.

This work has been financially supported by the Research Council of Finland under grants 342145, the Swedish Research Council grant 2016-00782, the Independent Research Fund Denmark grant 9040-00389B, the Wallenberg AI, Autonomous Systems and Software Program (WASP) funded by the Knut and Alice Wallenberg

Foundation, and the Fonds Wetenschappelijk Onderzoek – Vlaanderen (project G070521N).

Appendix A Formalization of the Proof Logging of SIS with the DPW

In this appendix, we provide formal details on the claims made in the main body of the paper. In the proofs, we follow the same notation. The formalization of the reasoning in the coarse convergence is discussed in Section 4.2, here we discuss the other phases.

A.1 Coarse Convergence

Our first proposition formalizes the wlog performed during the coarse convergence phase.

Proposition 3 (Proposition 2, restated)**.** *Assume the definition of z_k has been derived and a complete shadow circuit for $T = 0$ has been introduced. Furthermore assume the constraint*

$$O \geq 1 + k \cdot 2^P \tag{4}$$

has been derived. The constraint $\overline{z}_k \geq 1$ can be derived using redundance-based strengthening with witness

$$\omega = T \mapsto 0, Y \mapsto Y^{T=0}.$$

The notation for the witness in this proposition is a shorthand for the mapping that sends each variable t_i to 0 and every introduced circuit variable y to the corresponding shadow circuit variable $y^{T=0}$.

Proof. To verify this is indeed possible, we need to show that from

$$\mathcal{C} \cup \mathcal{D} \cup \{z_k \geq 1\}$$

we can derive the following constraints:

- $\overline{z}_k\restriction_\omega \geq 1$; in other words we need to show that $\overline{z}_k^{T=0} \geq 1$ holds. Recall that $z_k^{T=0}$ is defined by the reification

  $$\overline{z}_k^{T=0} \Leftrightarrow O - 0 \geq 1 + k \cdot 2^P.$$

  Adding up one direction of this definition to (4), immediately yields that $\overline{z}_k^{T=0} \geq 1$, as desired.

- $C\restriction_\omega$ for each $C \in \mathcal{C}$.
  - If C is a clause in the original input, $C\restriction_\omega = C$ and this is trivial.
  - If C is a previously derived solution-improving constraint, also $C\restriction_\omega = C$ (since ω does not touch any variable in O.

  - If C is a previously derived constraint of the form $\overline{z}_{k'} \geq 1$ with $k' < k$, this can either be derived analogously to $\overline{z}_k\restriction_\omega \geq 1$ or directly from the fact that the definitions of z_k and z'_k immediately imply that $z_k = 0$ implies that $z_{k'} = 0$.

- $O\restriction_\omega \geq O$; this is obvious since the variables in O are unaltered by ω. □

Remark 1. Proposition 3 assumes the existence of a constraint (4). It can be seen that this constraint is actually a (potentially weakened version of a) non-strict solution improving constraint $O \geq O\restriction_\alpha$ where α is a previously found solution. During the coarse convergence phase, this constraint can be obtained by weakening the solution-improving constraint.

At the end of the coarse convergence phase, also the unit clause $z_{k^*} \geq 1$ is derived. This requires no additional proof logging: this clause is obtained by running the SAT solver with the assumption that $z_{k^*} = 0$ and failing. Whenever this is the case; we know that $z_{k^*} \geq 1$ is internally derived by standard conflict analysis; hence this constraint is added to $\mathcal{D}$ without any additional effort.

A.2 Fine Convergence

As with the coarse convergence, the constraints derived during fine convergence that require a justification in the proof are the unit clauses added to the solver. Proving this relies again on redundance-based strengthening and a shadow circuit.

Proposition 4. *Assume $\overline{z}_{k^*-1} \geq 1$ has been derived. Let s be any number and assume a complete shadow circuit for $T = s - 1$ has been introduced. Furthermore assume the constraint*

$$O \geq s + (k^* - 1) \cdot 2^P \tag{5}$$

has been derived. The constraint $T \geq s - 1$ can be derived using redundance-based strengthening with witness

$$\omega = T \mapsto s, Y \mapsto Y^{T=s-1}.$$

Proof. As in the proof of Proposition 2, this yields several proof obligations. The only non-trivial ones are

- Previously derived constraints of this form $T \geq s' - 1$, but they are trivially satisfied under ω since $s \geq s'$.
- The unit clause $\overline{z}_{k^*-1} \geq 1\restriction_\omega$. In other words we need to show that $\overline{z}^{T=s-1}_{k^*-1}$ holds. Recall that $z^{T=s-1}_{k^*-1}$ is defined by the reification

$$\overline{z}^{T=s-1}_{k^*-1} \Leftrightarrow O - (s-1) \geq 1 + (k^* - 1) \cdot 2^P$$

  which simplifies to

$$\overline{z}^{T=s-1}_{k^*-1} \Leftrightarrow O - s \geq (k^* - 1) \cdot 2^P.$$

  Now (5) tells us precisely that the right-hand side of this equivalence is satisfied, hence a straightforward cutting planes derivation indeed allows us to conclude that $\overline{z}^{T=s}_{k^*-1} \geq 1$. □

Remark 2. Just like Proposition 2, also Proposition 4 does not make use of the model-improving constraint, but rather makes the assumption on O it uses explicit in (5). As before, this turns out to be useful when applying Proposition 4 in the context of stratification.

Proposition 4 will be applied when a solution α is found taking

$$s := O\restriction_\alpha -(k^* - 1) \cdot 2^P.$$

In this case, the solution-improving tells us that

$$O \geq O\restriction_\alpha +1 = s + (k^* - 1) \cdot 2^P + 1,$$

and (5) is indeed satisfied. Unit clauses are derived if for a certain j, $s \geq 2^P - 2^j + 1$. In this case, the derived constraint $T \geq s - 1$ guarantees that $T \geq 2^P - 2^j$, i.e., that all dominant bits of T up to j must be equal to one. This follows using reverse unit propagation or a straightforward cutting planes derivation.

A.3 Conclusion of Optimality

When the very last call to the SAT solver is unsatisfiable, we need to derive a contradiction in the proof, to complete the proof that the previously best found solution is optimal. We proceed as follows. First, we introduce a fresh variable, let us call it p using the reification

$$p \Leftrightarrow O \geq o^* + 1. \tag{6}$$

Our goal will be to show that p is false, which then allows us to conclude that the objective can no longer be improved, meaning we have indeed proven optimality. Recall that at this point, we have s defined as $s := o^* - (k^* - 1) \cdot 2^P$. The crucial step in our proof is showing that without loss of generality T can be set equal to s. We proceed as follows.

Proposition 5. *Assume $\overline{z}_{k^*-1} \geq 1$ and the definition of p have been derived. Furthermore suppose that a shadow circuit for $T = s$ has been introduced. Using redundance-based strengthening with witness*

$$\omega = T \mapsto s, Y \mapsto Y^{T=s}$$

we can derive the PB constraints representing

$$p \Rightarrow T = s, \tag{7}$$

i.e., in normalised form, the constraints

$$s \cdot \overline{p} + T \geq s, \text{ and} \tag{8}$$

$$(2^P - s - 2) \cdot \overline{p} + \sum_{j=0}^{P-1} 2^j \cdot \overline{T}_j \geq (2^P - 1) - s - 1. \tag{9}$$

Proof. The proof for the two constraints is similar. The only proof goal where they differ is showing that the constraint to-be-derived is satisfied under ω, but this is trivial since the witness sets T equal to s by construction.

For all the other proof goals, we can make use the negation of the constraint to be derived (the negation of (8) or of (9)). From this negation, we can directly derive $p \geq 1$. Adding this up to (one direction of (6) yields $O \geq o^* + 1$, i.e., that

$$O \geq s + (k^* - 1) \cdot 2^P + 1. \tag{10}$$

In other words, the conditions of 4 are satisfied. All the other proof obligations are the same as the ones in the proof of that proposition and hence, making use of (10), the proof proceeds identically to the proof of Proposition 4. □

In words, Proposition 5 tells us is that *if* the objective is strictly improving on the previously found best value, *then* we can set T equal to s without loss of generality. The SAT solver, however, has in its last call that yielded UNSAT already derived a clause telling us that at least one of the bits of T does not correspond to s. So we can now straightforwardly derive that $\overline{p} \geq 1$ and hence that $O \leq o^*$, which is what we needed for concluding optimality.

Appendix B Proof Logging of Additional Techniques Implemented in Pacose

We detail some of the additional search techniques implemented in and how we proof log them. As a minor point, we note for completeness that in addition to the gcd-based criterion described in Section 3.4, PACOSE attempts to find more partitions of the objective during stratification via exhaustive search, as illustrated by the following example:

Example 3. Consider the objective $O := 14x_1 + 9x_2 + 5x_3 + 2x_4 + 1x_5 + 1x_6$ and the partition $H = \{1, 2, 3\}$ and $L = \{4, 5, 6\}$. According to the gcd-based criterion from Section 3.4, this partition is not viable due to the gcd not aligning with any single divisor that groups the weights cohesively. However, this partition still validly separates the weights of x_1 to x_6 through an alternative method: Define L_C as the set containing all possible summed combinations of weights from L: $L_C := 5, 9, 14, 5 + 9, 5 + 14, 9 + 14, 5 + 9 + 14$. To validate this partitioning, ensure that the total weight W_L from L is at most the difference between any two sums in L_C. This ensures that L forms a consistent grouping, as there is no weight combination of L invalidating a prior result of solving H.

A more in-depth explanation together with a proof can be found in [PRB21]. While certifying the exhaustive search remains interesting future work, we note that it did not result in additional partitions on any of the benchmarks in our evaluation, nor on the weighted instances of the 2019 and 2020 MaxSAT Evaluation.

We would like to mention that a naive approach to certify the exhaustive search would be to derive the desired constraint $O_H \geq O_H\restriction_\alpha$ from the weakened constraint $O_H \geq O\restriction_\alpha - W_L + 1$ using redundance-based strengthening with an empty witness.

As $O_H\restriction_\alpha$ is the sum of a subset of the coefficients in O_H and the distance between any two sums is at least W_L, the negation $O_H < O_H\restriction_\alpha$ of the desired constraint can only be satisfied if the sum of true literals in O_H is at most $O_H\restriction_\alpha - W_L$. As $O\restriction_\alpha \geq O_H\restriction_\alpha$, the weakened constraint can only be satisfied if the sum of true literals in O_H is at least $O_H\restriction_\alpha - W_L + 1$. Hence, there exists no assignment to the variables in O_H for which both constraints are satisfied. To show this we can iterate through every possible assignment α of the variables in O_H and derive the clause excluding this assignment by reverse unit propagation. This step works, as reverse unit propagation for this clause assigns all variables in O_H, which will falsify either the negated constraint or the weakened constraint by the arguments above. Resolving all the clauses will result in a contradiction that proves that $O_H \geq O_H\restriction_\alpha$ is implied.

B.1 TrimMaxSAT

TrimMaxSAT [PRB21] is a preprocessing technique applied before the main SIS algorithm in order to decrease the number of literals in the objective that need to be encoded by the DPW and to get a good initial value of the objective. TrimMaxSAT heuristically splits the variables in the objective into partitions and queries the SAT solver for a solution that assigns at least one of the literals in each partition to 1. If such an assignment is found, the objective variables set to 1 are removed from consideration and the number of partitions are decreased. If the partition size is 1 and the SAT solver reports UNSAT, all remaining literals are fixed to 0 for the rest of the search. In other words TrimMaxSAT aims to find objective literals whose negation is implied by the constraints in the formula and fix their value, thus conceptually decreasing the size of the objective under consideration and–as a consequence–also the size of the DPW encoding built over it.

In more detail, assume $\mathcal{L}$ contains the set of objective variables that have not been set to 1 in any solutions found so far during TrimMaxSAT. During an iteration of TrimMaxSAT, $\mathcal{L}$ is partitioned into m subsets $\mathcal{L}^i$ for $i = 1, \ldots, m$. A new variable r is introduced and the clauses $r \Rightarrow (\sum_{\ell \in \mathcal{L}^i} \ell \geq 1)$ for every $i = 1, \ldots, m$ are added to the SAT solver and the proof via redundance-based strengthening to the core set. The SAT solver is then queried under the assumption that r is true. If the result is SAT, the literals in $\mathcal{L}$ assigned to 1 in the obtained solution are removed from the set under consideration and the unit clause $\overline{r} \geq 1$ is added to the solver such that the SAT solver can remove the clauses of the form $r \Rightarrow (\sum_{\ell \in \mathcal{L}^i} \ell \geq 1)$. This unit clause can be derived by redundance-based strengthening with witness $\omega = r \mapsto 0$. If, on the other hand, the result is UNSAT, the unit clause $\overline{r} \geq 1$ is added to the SAT solver and the SAT solver can simplify its clause database. This clause is derived by standard cutting planes reasoning in the conflict analysis by the SAT solver and is therefore added to the derived set in the proof. If in this case $m = 1$, we can also conclude that all literals $\ell \in \mathcal{L}$ are implied to be false. Hence, the solver learns the unit clauses $\overline{\ell} \geq 1$. In order to derive $\overline{\ell} \geq 1$ for each $\ell \in \mathcal{L}^i$, we first introduce the second part of the reification $r \Leftarrow (\sum_{\ell \in \mathcal{L}^i} \ell \geq 1)$ using the redundance rule with witness $r \mapsto 1$ and then use cutting planes reasoning to derive that since r is false, all literals in $\mathcal{L}^i$ must be false. Interestingly, thanks to the use of strengthening-to-core, the unit clause $\overline{r} \geq 1$ derived earlier does not interfere with the derivation of the second direction of the reification.

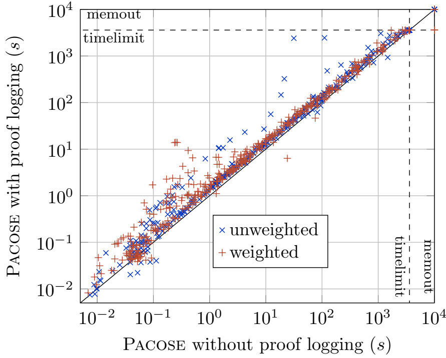


Figure 4: *Proof logging overhead for PACOSE using the binary adder encoding.*

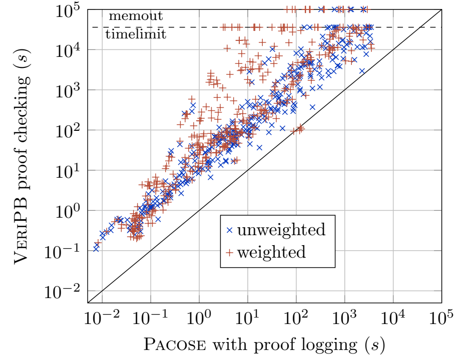


Figure 5: *PACOSE vs. VERIPB running time using binary adder encoding.*

B.2 Hardening

Hardening refers to the addition of the unit clause l_i for an objective literal l_i if the currently best known solution o^* is larger than the sum of all weights in O excluding w_i. In the proof, the unit clause l_i can be derived easily from the solution-improving constraint and the objective reformulation rule can be used to replace l_i by the constant w_i in the objective.

Appendix C Additional Experimental Evaluation

In this appendix, we present some additional experimental analysis with data and plots to give some further insights into proof logging for PACOSE. In Section C.1, we present results for the binary adder encoding that is also used in PACOSE and how detail how well proof logging performs for PACOSE when it heuristically selects the encoding. We present data for an additional approach that uses assumptions instead of unit clauses for fixing variables in the coarse convergence in Section C.2. To better understand the proof logging overhead in PACOSE, we have a deeper look at some additional data for the proof logging process in Section C.3.

C.1 Binary Adder Encoding and Encoding Selection Heuristic

PACOSE also uses the binary adder encoding [War98] instead of the DPW encoding. A comparison between these two encodings is beyond the scope of this paper, but as we implemented proof logging for both encodings, we can also have a look at the data for the binary adder encoding. A comparison of solving with and without proof logging for this encoding can be found in Figure 4. With proof logging for the binary adder encoding 722 instances could be solved within the resource limits, which are 6 fewer instances than without proof logging. This also demonstrates that the heuristic for selecting the encoding works, as the number of solved instances for the heuristic is bigger than for any of the two encodings on

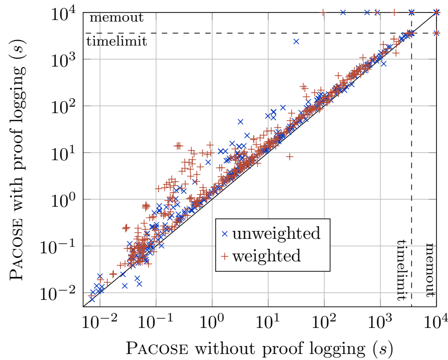


Figure 6: *Proof logging overhead for* PACOSE *using heuristic encoding selection.*

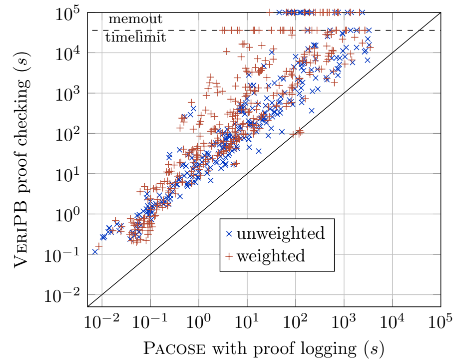


Figure 7: PACOSE *vs.* VERIPB *running time using heuristic encoding selection.*

their own. In the mean, PACOSE with proof logging is 1.63× slower than without proof logging. This overhead is smaller than for the DPW encoding, which lead to the conclusion that more work is required to certify the DPW encoding compared to the binary adder encoding.

Out of the 722 instances that were solved with the binary adder encoding, 658 instances were successfully checked by VERIPB within the resource limits. In Figure 5, the running time of PACOSE is compared to that of VERIPB. In the mean, VERIPB is 21.1× slower than PACOSE for solving the instance with proof logging, which is similar to the DPW encoding. This could mean that the bottleneck for checking the proofs is the implementation of the checker.

Using the default settings, PACOSE heuristically selects between the DPW and binary adder encoding. A plot comparing PACOSE with and without proof logging in the default settings in Figure 6 and a plot comparing PACOSE with proof logging with VERIPB for checking the proof in Figure 7. With this heuristic activated, 698 instances are solved within the resource limits with proof logging enabled and 707 instances without. PACOSE with proof logging is 1.83× slower in the mean than PACOSE without proof logging. Checking the proof with VERIPB is 21.8× slower than running PACOSE with proof logging in the mean.

C.2 Coarse Convergence with Assumptions Instead of Unit Clauses

An alternative approach for representing the information that output variables of the DPW encoding are fixed to a value in the coarse convergence is to use additional assumptions for the SAT solver instead of unit clauses. As we need a shadow circuit to derive each unit clause, we could reduce the number of shadow circuits by using assumptions. The idea is that we add the variable fixing to the assumptions for all future calls to the SAT solver. This approach is supported in PACOSE, and we ran additional experiments using this approach.

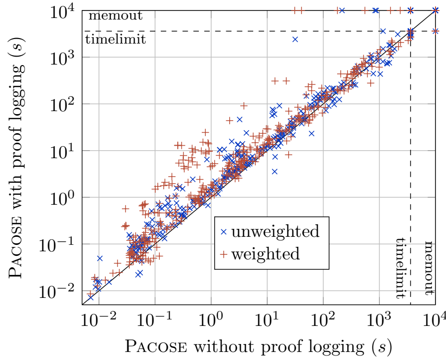


Figure 8: *Proof logging overhead for PACOSE using DPW encoding and assumptions.*

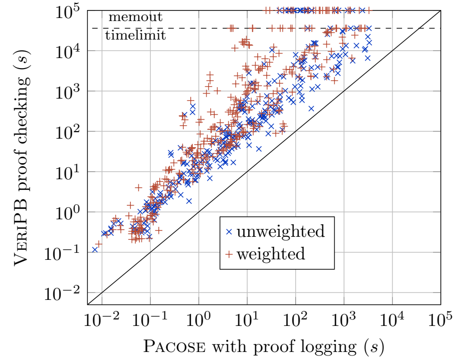


Figure 9: *PACOSE vs. VERIPB running time using DPW encoding and assumptions.*

The following data always use assumptions instead of unit clauses for fixing variables. In Figure 8, PACOSE with proof logging is compared to PACOSE without proof logging. Using assumptions PACOSE with proof logging could solve 666 instances, which is 10 fewer instances than without proof logging. PACOSE with proof logging is 1.81× slower than without proof logging in the mean. This is very similar to PACOSE with the DPW encoding where the variables are fixed by unit clauses and introducing shadow circuits. In the mean, the proof checking is 22.2× slower than solving the instance with proof logging.

It can be concluded that this alternative approach of fixing variables by adding assumptions is about as good as doing the fixing by unit clauses. Hence, it could be that introducing additional shadow circuits for deriving the unit clauses does not slow down the solving a lot, or it is a coincidence that the performance gains are countered by the additional work required for keeping track of the assumptions.

C.3 Proof Logging Overhead Analysis

To get a better understanding of the 1.93× slowdown of PACOSE with proof logging compared to without proof logging, we investigate different causes for the extra running time with proof logging. The idea for doing so is to get insights into how to improve the running time of the solvers.

The expectation is that the proof size scales linearly with the running time of the solver. It would be interesting to look into the instances where this is not the case and if there is a correlation with the solving overhead. We can illustrate this by plotting the solving time against the proof size and colour the marks depending on the overhead as it is done in Figure 10 for the DPW encoding and in Figure 11 for the binary adder encoding. We added a diagonal line representing linear scaling of proof size with running time for better orientation, which is not related to the data at all. It can be seen that for the instances that have a proof size that is significantly bigger than expected, the overhead also seems to increase similarly. To confirm

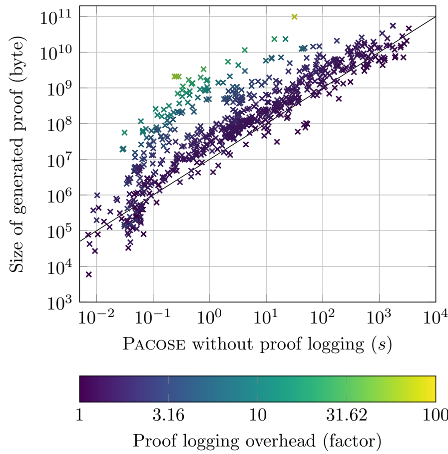


Figure 10: *Solving time vs. proof size vs. solving overhead for proof logging for the DPW encoding.*

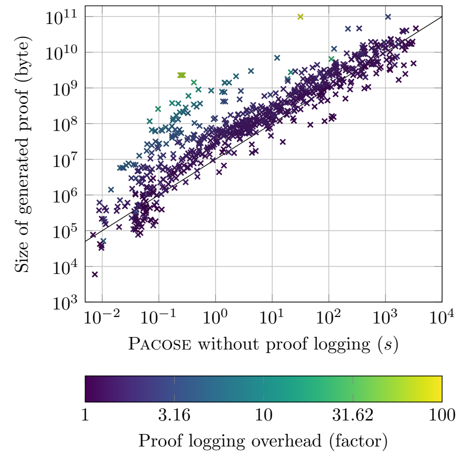


Figure 11: *Solving time vs. proof size vs. solving overhead for proof logging for the binary adder encoding.*

this observation, we compute the correlation of the proof logging overhead and the proof size divided by the solving time. For the DPW encoding we have a correlation of 0.92 and for the binary adder encoding we have a correlation of 0.88, which shows that the two parameters are highly correlated. This mean that the slowdown is due to proof being larger than expected for some instances.

We can conclude with some ideas to improve the performance of proof logging in Pacose. First, the performance can be improved by engineering better data structures to handle the proof logging to increase the speed for writing the proof. This idea only works if we have not reached the maximum persistent disk write speed, which is not the case for our experiments. Second, the proof could be done in a smarter way to reduce the size of the proof, where slow parts of the proof logging could be identified by profiling. Considering that we also have a 1.63× slowdown for the binary adder encoding, the slowdown is not purely caused by the shadow circuits, as they are not used for this encoding.

References

[ABM+11] Eyad Alkassar, Sascha Böhme, Kurt Mehlhorn, Christine Rizkallah, and Pascal Schweitzer. An introduction to certifying algorithms. *it - Information Technology Methoden und innovative Anwendungen der Informatik und Informationstechnik*, 53(6):287–293, December 2011.

[ALM09] Josep Argelich, Inês Lynce, and João P. Marques-Silva. On solving Boolean multilevel optimization problems. In *Proceedings of the 21st International Joint Conference on Artificial Intelligence (IJCAI '09)*, pages 393–398, July 2009.

[BB03] Olivier Bailleux and Yacine Boufkhad. Efficient CNF encoding of Boolean cardinality constraints. In *Proceedings of the 9th International Conference on Principles and Practice of Constraint Programming (CP '03)*, volume 2833 of *Lecture Notes in Computer Science*, pages 108–122. Springer, September 2003.

[BB09] Robert Brummayer and Armin Biere. Fuzzing and delta-debugging SMT solvers. In *Proceedings of the 7th International Workshop on Satisfiability Modulo Theories (SMT '09)*, pages 1–5, August 2009.

[BBN+23] Jeremias Berg, Bart Bogaerts, Jakob Nordström, Andy Oertel, and Dieter Vandesande. Certified core-guided MaxSAT solving. In *Proceedings of the 29th International Conference on Automated Deduction (CADE-29)*, volume 14132 of *Lecture Notes in Computer Science*, pages 1–22. Springer, July 2023.

[BBN+24] Jeremias Berg, Bart Bogaerts, Jakob Nordström, Andy Oertel, Tobias Paxian, and Dieter Vandesande. Experimental Repository for "Certifying Without Loss of Generality Reasoning in Solution-Improving Maximum Satisfiability", June 2024.

[BBR09] Olivier Bailleux, Yacine Boufkhad, and Olivier Roussel. New encodings of pseudo-Boolean constraints into CNF. In *Proceedings of the 12th International Conference on Theory and Applications of Satisfiability Testing (SAT '09)*, volume 5584 of *Lecture Notes in Computer Science*, pages 181–194. Springer, June 2009.

[BCH21] Seulkee Baek, Mario Carneiro, and Marijn J. H. Heule. A flexible proof format for SAT solver-elaborator communication. In *Proceedings of the 27th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS '21)*, volume 12651 of *Lecture Notes in Computer Science*, pages 59–75. Springer, March-April 2021.

[BGMN23] Bart Bogaerts, Stephan Gocht, Ciaran McCreesh, and Jakob Nordström. Certified dominance and symmetry breaking for combinatorial optimisation. *Journal of Artificial Intelligence Research*, 77:1539–1589, August 2023. Preliminary version in *AAAI '22*.

[BHvMW21] Armin Biere, Marijn J. H. Heule, Hans van Maaren, and Toby Walsh, editors. *Handbook of Satisfiability*, volume 336 of *Frontiers in Artificial Intelligence and Applications*. IOS Press, 2nd edition, February 2021.

[Bie06] Armin Biere. Tracecheck. `http://fmv.jku.at/tracecheck/`, 2006.

[BJM21] Fahiem Bacchus, Matti Järvisalo, and Ruben Martins. Maximum satisfiabiliy. In Biere et al. [BHvMW21], chapter 24, pages 929–991.

[BLB10] Robert Brummayer, Florian Lonsing, and Armin Biere. Automated testing and debugging of SAT and QBF solvers. In *Proceedings of the 13th International Conference on Theory and Applications of Satisfiability Testing (SAT '10)*, volume 6175 of *Lecture Notes in Computer Science*, pages 44–57. Springer, July 2010.

[BLM07] Maria Luisa Bonet, Jordi Levy, and Felip Manyà. Resolution for Max-SAT. *Artificial Intelligence*, 171(8-9):606–618, June 2007. Extended version of paper in *SAT '06*.

[BMM+23] Bart Bogaerts, Ciaran McCreesh, Magnus O. Myreen, Jakob Nordström, Andy Oertel, and Yong Kiam Tan. Documentation of VeriPB and CakePB for the SAT competition 2023. Available at `https://satcompetition.github.io/2023/checkers.html`, March 2023.

[BN21] Samuel R. Buss and Jakob Nordström. Proof complexity and SAT solving. In Biere et al. [BHvMW21], chapter 7, pages 233–350.

[BRK+22] Haniel Barbosa, Andrew Reynolds, Gereon Kremer, Hanna Lachnitt, Aina Niemetz, Andres Nötzli, Alex Ozdemir, Mathias Preiner, Arjun Viswanathan, Scott Viteri, Yoni Zohar, Cesare Tinelli, and Clark Barrett. Flexible proof production in an industrial-strength SMT solver. In *Proceedings of the 11th International Joint Conference on Automated Reasoning (IJCAR '22)*, volume 13385 of *Lecture Notes in Computer Science*, pages 15–35. Springer, August 2022.

[CCT87] William Cook, Collette Rene Coullard, and György Turán. On the complexity of cutting-plane proofs. *Discrete Applied Mathematics*, 18(1):25–38, November 1987.

[CKSW13] William Cook, Thorsten Koch, Daniel E. Steffy, and Kati Wolter. A hybrid branch-and-bound approach for exact rational mixed-integer programming. *Mathematical Programming Computation*, 5(3):305–344, September 2013.

[DB13] Jessica Davies and Fahiem Bacchus. Exploiting the power of MIP solvers in MAXSAT. In *Proceedings of the 16th International Conference on Theory and Applications of Satisfiability Testing (SAT '13)*, volume 7962 of *Lecture Notes in Computer Science*, pages 166–181. Springer, July 2013.

[DEGH23] Jasper van Doornmalen, Leon Eifler, Ambros Gleixner, and Christopher Hojny. A proof system for certifying symmetry and optimality reasoning in integer programming. Technical Report 2311.03877, arXiv.org, November 2023.

[EG23] Leon Eifler and Ambros Gleixner. A computational status update for exact rational mixed integer programming. *Mathematical Programming*, 197(2):793–812, February 2023.

[EH20] Salomé Eriksson and Malte Helmert. Certified unsolvability for SAT planning with property directed reachability. In *Proceedings of the 30th International Conference on Automated Planning and Scheduling*, pages 90–100, October 2020.

[ERH17] Salomé Eriksson, Gabriele Röger, and Malte Helmert. Unsolvability certificates for classical planning. In *Proceedings of the 27th International*

Conference on Automated Planning and Scheduling (ICAPS '17), pages 88–97, June 2017.

[ERH18] Salomé Eriksson, Gabriele Röger, and Malte Helmert. A proof system for unsolvable planning tasks. In *Proceedings of the 28th International Conference on Automated Planning and Scheduling (ICAPS '18)*, pages 65–73, June 2018.

[ES03] Niklas Eén and Niklas Sörensson. Temporal induction by incremental SAT solving. In *Proceedings of the 1st International Workshop on Bounded Model Checking (BMC '03)*, volume 89 of *Electronic Notes in Theoretical Computer Science*, pages 543–560, July 2003.

[ES06] Niklas Eén and Niklas Sörensson. Translating pseudo-Boolean constraints into SAT. *Journal on Satisfiability, Boolean Modeling and Computation*, 2(1-4):1–26, March 2006.

[Fle20] Mathias Fleury. *Formalization of Logical Calculi in Isabelle/HOL*. PhD thesis, Universität des Saarlandes, 2020. Available at `https://publikationen.sulb.uni-saarland.de/handle/20.500.11880/28722`.

[FM06] Zhaohui Fu and Sharad Malik. On solving the partial MAX-SAT problem. In *Proceedings of the 9th International Conference on Theory and Applications of Satisfiability Testing (SAT '06)*, volume 4121 of *Lecture Notes in Computer Science*, pages 252–265. Springer, August 2006.

[GMM+20] Stephan Gocht, Ross McBride, Ciaran McCreesh, Jakob Nordström, Patrick Prosser, and James Trimble. Certifying solvers for clique and maximum common (connected) subgraph problems. In *Proceedings of the 26th International Conference on Principles and Practice of Constraint Programming (CP '20)*, volume 12333 of *Lecture Notes in Computer Science*, pages 338–357. Springer, September 2020.

[GMM+24] Stephan Gocht, Ciaran McCreesh, Magnus O. Myreen, Jakob Nordström, Andy Oertel, and Yong Kiam Tan. End-to-end verification for subgraph solving. In *Proceedings of the 368h AAAI Conference on Artificial Intelligence (AAAI '24)*, pages 8038–8047, February 2024.

[GMNO22] Stephan Gocht, Ruben Martins, Jakob Nordström, and Andy Oertel. Certified CNF translations for pseudo-Boolean solving. In *Proceedings of the 25th International Conference on Theory and Applications of Satisfiability Testing (SAT '22)*, volume 236 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 16:1–16:25, August 2022.

[GN03] Evgueni Goldberg and Yakov Novikov. Verification of proofs of unsatisfiability for CNF formulas. In *Proceedings of the Conference on Design, Automation and Test in Europe (DATE '03)*, pages 886–891, March 2003.

[GN21] Stephan Gocht and Jakob Nordström. Certifying parity reasoning efficiently using pseudo-Boolean proofs. In *Proceedings of the 35th AAAI Conference on Artificial Intelligence (AAAI '21)*, pages 3768–3777, February 2021.

[GSD19] Xavier Gillard, Pierre Schaus, and Yves Deville. SolverCheck: Declarative testing of constraints. In *Proceedings of the 25th International Conference on Principles and Practice of Constraint Programming (CP '19)*, volume 11802 of *Lecture Notes in Computer Science*, pages 565–582. Springer, October 2019.

[HOGN24] Alexander Hoen, Andy Oertel, Ambros Gleixner, and Jakob Nordström. Certifying MIP-based presolve reductions for 0–1 integer linear programs. In *Proceedings of the 21st International Conference on the Integration of Constraint Programming, Artificial Intelligence, and Operations Research (CPAIOR '24)*, volume 14742 of *Lecture Notes in Computer Science*, pages 310–328. Springer, May 2024.

[IOT+24] Hannes Ihalainen, Andy Oertel, Yong Kiam Tan, Jeremias Berg, Matti Järvisalo, Magnus O. Myreen, and Jakob Nordström. Certified MaxSAT preprocessing. In *Proceedings of the 12th International Joint Conference on Automated Reasoning (IJCAR '24)*, volume 14739 of *Lecture Notes in Computer Science*, pages 396–418. Springer, July 2024.

[JHB12] Matti Järvisalo, Marijn J. H. Heule, and Armin Biere. Inprocessing rules. In *Proceedings of the 6th International Joint Conference on Automated Reasoning (IJCAR '12)*, volume 7364 of *Lecture Notes in Computer Science*, pages 355–370. Springer, June 2012.

[JMM15] Saurabh Joshi, Ruben Martins, and Vasco M. Manquinho. Generalized totalizer encoding for pseudo-Boolean constraints. In *Proceedings of the 21st International Conference on Principles and Practice of Constraint Programming (CP '15)*, volume 9255 of *Lecture Notes in Computer Science*, pages 200–209. Springer, August-September 2015.

[KM21] Sonja Kraiczy and Ciaran McCreesh. Solving graph homomorphism and subgraph isomorphism problems faster through clique neighbourhood constraints. In *Proceedings of the 30th International Joint Conference on Artificial Intelligence (IJCAI '21)*, pages 1396–1402, August 2021.

[KP19] Michal Karpinski and Marek Piotrów. Encoding cardinality constraints using multiway merge selection networks. *Constraints*, 24(3–4):234–251, October 2019.

[LBJ20] Marcus Leivo, Jeremias Berg, and Matti Järvisalo. Preprocessing in incomplete MaxSAT solving. In *Proceedings of the 24th European Conference on Artificial Intelligence (ECAI '20)*, volume 325 of *Frontiers in Artificial Intelligence and Applications*, pages 347–354, August-September 2020.

[LM21] Chu Min Li and Felip Manyà. MaxSAT, hard and soft constraints. In Biere et al. [BHvMW21], chapter 23, pages 903–927.

[LNOR11] Javier Larrosa, Robert Nieuwenhuis, Albert Oliveras, and Enric Rodríguez-Carbonell. A framework for certified Boolean branch-and-bound optimization. *Journal of Automated Reasoning*, 46(1):81–102, January 2011.

[Max] MaxSAT evaluations: Evaluating the state of the art in maximum satisfiability solver technology. `https://maxsat-evaluations.github.io/`.

[Max23] MaxSAT evaluation 2023. `https://maxsat-evaluations.github.io/2023`, July 2023.

[MM11] António Morgado and João P. Marques-Silva. On validating Boolean optimizers. In *Proceedings of the 23rd IEEE International Conference on Tools with Artificial Intelligence (ICTAI '12)*, pages 924–926, November 2011.

[MMNS11] Ross M. McConnell, Kurt Mehlhorn, Stefan Näher, and Pascal Schweitzer. Certifying algorithms. *Computer Science Review*, 5(2):119–161, May 2011.

[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 *Proceedings of the 37th Annual German Conference on Artificial Intelligence (KI '14)*, volume 8736 of *Lecture Notes in Computer Science*, pages 123–134. Springer, September 2014.

[PB23] Tobias Paxian and Armin Biere. Uncovering and classifying bugs in MaxSAT solvers through fuzzing and delta debugging. In *Proceedings of the 14th International Workshop on Pragmatics of SAT*, volume 3545 of *CEUR Workshop Proceedings*, pages 59–71. CEUR-WS.org, July 2023.

[PCH20] Matthieu Py, Mohamed Sami Cherif, and Djamal Habet. Towards bridging the gap between SAT and Max-SAT refutations. In *Proceedings of the 32nd IEEE International Conference on Tools with Artificial Intelligence (ICTAI '20)*, pages 137–144, November 2020.

[PCH21] Matthieu Py, Mohamed Sami Cherif, and Djamal Habet. A proof builder for Max-SAT. In *Proceedings of the 24th International Conference on Theory and Applications of Satisfiability Testing (SAT '21)*, volume 12831 of *Lecture Notes in Computer Science*, pages 488–498. Springer, July 2021.

[PCH22] Matthieu Py, Mohamed Sami Cherif, and Djamal Habet. Proofs and certificates for Max-SAT. *Journal of Artificial Intelligence Research*, 75:1373–1400, December 2022.

[PRB18] Tobias Paxian, Sven Reimer, and Bernd Becker. Dynamic polynomial watchdog encoding for solving weighted MaxSAT. In *Proceedings of the 21st International Conference on Theory and Applications of Satisfiability Testing (SAT '18)*, volume 10929 of *Lecture Notes in Computer Science*, pages 37–53. Springer, July 2018.

[PRB21] Tobias Paxian, Pascal Raiola, and Bernd Becker. On preprocessing for weighted MaxSAT. In *Proceedings of the 22nd International Conference on Verification, Model Checking, and Abstract Interpretation (VMCAI '21)*, volume 12597 of *Lecture Notes in Computer Science*, pages 556–577. Springer, January 2021.

[Rög17] Gabriele Röger. Towards certified unsolvability in classical planning. In *Proceedings of the 26th International Joint Conference on Artificial Intelligence (IJCAI '17)*, pages 5141–5145, August 2017.

[SFBF21] Hans-Jörg Schurr, Mathias Fleury, Haniel Barbosa, and Pascal Fontaine. Alethe: Towards a generic SMT proof format (extended abstract). In *Proceedings of the 7th Workshop on Proof eXchange for Theorem Proving (PxTP '21*, volume 336 of *Electronic Proceedings in Theoretical Computer Science*, pages 49–54, July 2021.

[Sin05] Carsten Sinz. Towards an optimal CNF encoding of Boolean cardinality constraints. In *Proceedings of the 11th International Conference on Principles and Practice of Constraint Programming (CP '05)*, volume 3709 of *Lecture Notes in Computer Science*, pages 827–831. Springer, October 2005.

[Van23] Dieter Vandesande. Towards certified MaxSAT solving: Certified MaxSAT solving with SAT oracles and encodings of pseudo-Boolean constraints. Master's thesis, Vrije Universiteit Brussel (VUB), 2023.

[VDB22] Dieter Vandesande, Wolf De Wulf, and Bart Bogaerts. QMaxSATpb: A certified MaxSAT solver. In *Proceedings of the 16th International Conference on Logic Programming and Non-monotonic Reasoning (LPNMR '22)*, volume 13416 of *Lecture Notes in Computer Science*, pages 429–442. Springer, September 2022.

[Ver] VeriPB: Verifier for pseudo-Boolean proofs. `https://gitlab.com/MIAOresearch/software/VeriPB`.

[War98] Joost P. Warners. A linear-time transformation of linear inequalities into conjunctive normal form. *Information Processing Letters*, 68(2):63–69, October 1998.

[WHH14] Nathan Wetzler, Marijn J. H. Heule, and Warren A. Hunt Jr. DRAT-trim: Efficient checking and trimming using expressive clausal proofs. In *Proceedings of the 17th International Conference on Theory and Applications of Satisfiability Testing (SAT '14)*, volume 8561 of *Lecture Notes in Computer Science*, pages 422–429. Springer, July 2014.

Certifying MIP-Based Presolve Reductions for 0–1 Integer Linear Programs

Abstract

It is well known that reformulating the original problem can be crucial for the performance of mixed-integer programming (MIP) solvers. To ensure correctness, all transformations must preserve the feasibility status and optimal value of the problem, but there is currently no established methodology to express and verify the equivalence of two mixed-integer programs. In this work, we take a first step in this direction by showing how the correctness of MIP presolve reductions on 0–1 integer linear programs can be certified by using (and suitably extending) the VeriPB tool for pseudo-Boolean proof logging. Our experimental evaluation on both decision and optimization instances demonstrates the computational viability of the approach and leads to suggestions for future revisions of the proof format that will help to reduce the verbosity of the certificates and to accelerate the certification and verification process further.

1 Introduction

Boolean satisfiability solving (SAT) and *mixed-integer programming (MIP)* are two computational paradigms in which surprisingly mature and powerful solvers have been developed over the last decades. Today such solvers are routinely used to solve large-scale problems in practice despite the fact that these problems are *NP*-hard. Both SAT and MIP solvers typically start by trying to simplify the input problem before feeding it to the main solver algorithm, a process known as *presolving* in MIP and *preprocessing* in SAT. This can involve, e.g., fixing variables to values, strengthening constraints, removing constraints, or adding new constraints to break symmetries. Such techniques are very important for SAT

Alexander Hoen, Andy Oertel, Ambros Gleixner, and Jakob Nordström. "Certifying MIP-Based Presolve Reductions for 0–1 Integer Linear Programs". In *Proceedings of the 21st International Conference on the Integration of Constraint Programming, Artificial Intelligence, and Operations Research (CPAIOR '24)*, volume 14742 of *Lecture Notes in Computer Science*, pages 310–328. Springer, May 2024.

solver performance [BJK21], and for MIP solvers they often play a decisive role in whether a problem instance can be solved or not, regardless of whether the solver uses floating-point [ABG+19] or exact rational arithmetic [EG22].

The impressive performance gains for modern combinatorial solvers come at the price of ever-increasing complexity, which makes these tools very hard to debug. It is well documented that even state-of-the-art solvers in many paradigms, not just SAT and MIP, suffer from errors such as mistakenly claiming infeasibility or optimality, or even returning "solutions" that are infeasible [AGJ+18, CKSW13a, GSD19, Klo14, Ste11]. During the last decade, the SAT community has dealt with this problem in a remarkably successful way by requiring that solvers should use *proof logging*, i.e., produce machine-verifiable certificates of correctness for their computations that can be verified by a stand-alone proof checker. A number of proof formats have been developed, such as DRAT [HHW13a, HHW13b, WHH14], GRIT [CMS17], and LRAT [CHH+17], which are used to certify the whole solving process including preprocessing.

Achieving something similar in a general MIP setting is much more challenging, amongst others because of the presence of continuous and general integer variables, which may even have unbounded domains. For numerically exact MIP solvers [CKSW13b, EG22, EG24] the proof format VIPR [CGS17] has been introduced, but it currently only allows verification of feasibility-based reasoning, which must preserve all feasible solutions. In particular, it does not support the verification of dual presolving techniques that may exclude feasible solutions as long as one optimal solution remains. This means that while exact MIP solvers could in principle generate a certificate for the main solving process, such a certificate would only establish correctness under the assumption that all the presolving steps were valid, as, e.g., in [EG22]. And, unfortunately, the proof logging techniques for SAT preprocessing cannot be used to address this problem, since they can only reason about clausal constraints.

Our contribution in this work is to take a first step towards verification of the full MIP solving process by demonstrating how pseudo-Boolean proof logging with VeriPB can be used to produce certificates of correctness for a wide range of MIP presolving techniques for 0–1 integer linear programs (ILPs). VeriPB is quite a versatile tool in that it has previously been employed for certification of, e.g., advanced SAT solving techniques [BGMN23, GN21], SAT-based optimization (MaxSAT) [BBN+23, VDB22], subgraph solving [GMM+20, GMN20], and constraint programming [GMN22, MM23]. However, to the best of our knowledge this is the first time the tool has been used to prove the correctness of reformulations of optimization problems, and this presents new challenges. In particular, the proof system turns out not to be well suited for problem reformulations with frequent changes to the objective function, and therefore we introduce a new rule for objective function updates.

Our computational experiments confirm that this approach to certifying presolve reductions is computationally viable and the overhead for certification aligns with what is known from the literature for certifying problem transformations in other contexts [GMNO22]. The analysis of the results reveals new insights into performance bottlenecks, and these insights directly translate to possible revisions

of the proof logging format that would be valuable to address in order to decrease the size of the generated proofs and speed up proof verification.

We would like to note that, while our current methods are only applicable to 0–1 ILPs, this covers already a large and important class of MIPs. In particular, there are applications where the exact and verified solution of 0–1 ILPs is highly relevant, see [Ach07, EGP22, SBD19] for some examples.

The rest of this paper is organized as follows. After presenting pseudo-Boolean proof logging and VeriPB in Sec. 2, we demonstrate in Sec. 3 how to produce VeriPB certificates for MIP presolving on 0–1 ILPs. In Sec. 4 we report results of an experimental evaluation, and we conclude in Sec. 5 with a summary and discussion of future work.

2 Pseudo-Boolean Proof Logging with VeriPB

We start by reviewing pseudo-Boolean reasoning in Sec. 2.1, and then explain our extension to deal with objective function updates in Sec. 2.2. In order to make the concept of proof logging more concrete, we conclude this section by providing, in Tab. 1, a few examples of how the derivation rules explained below are encoded in VeriPB syntax. For space reasons, this list does not include examples of subproofs that may be necessary for some derivations that cannot be proven automatically by VeriPB. Further details on practical aspects and implementation of pseudo-Boolean proof logging can be found in the software repository of VeriPB [GO23].

2.1 Pseudo-Boolean Reasoning with the Cutting Planes Method

Our treatment of this material will by necessity be somewhat terse—we refer the reader to [BN21] for more information about the cutting planes method and to [BGMN23, GMNO22] for detailed information about the VeriPB proof system and format.

We write x to denote a $\{0, 1\}$-valued variable and $\overline{x}$ as a shorthand for $1 - x$, and write ℓ to denote such *positive* and *negative literals*, respectively. By a *pseudo-Boolean (PB) constraint* we mean a 0–1 linear inequality $\sum_j a_j \ell_j \geq b$, where when convenient we can assume all literals ℓ_j to refer to distinct variables and all a_j and b to be non-negative (so-called *normalized form*). A *pseudo-Boolean formula* is just another name for a 0–1 integer linear program. For optimization problems we also have an objective function $f = \sum_j c_j x_j$ that should be minimized (and f can be negated to represent a maximization problem).

The foundation of VeriPB is the *cutting planes* proof system [CCT87]. At the start of the proof, the set of *core constraints* $\mathcal{C}$ are initialized as the 0–1 linear inequalities in the problem instance. Any constraints derived as described below are placed in the set of *derived constraints* $\mathcal{D}$, from where they can later be moved to $\mathcal{C}$ (but not vice versa). Loosely speaking, VeriPB proofs maintain the invariant that the optimal value of any solution to $\mathcal{C}$ and to the original input problem is the same. New constraints can be derived from $\mathcal{C} \cup \mathcal{D}$ by performing *addition* of two constraints or *multiplication* of a constraint by a positive integer, and *literal axioms* $\ell \geq 0$ can be used at any time. Additionally, for a constraint $\sum_j a_j \ell_j \geq b$

written in normalized form we can apply *division* by a positive integer d followed by rounding up to obtain $\sum_j \lceil a_j/d \rceil \ell_j \geq \lceil b/d \rceil$, and *saturation* can be applied to yield $\sum_j \min\{a_j, b\} \cdot \ell_j \geq b$.

For a PB constraint $C \doteq \sum_j a_j \ell_j \geq b$ (where we use $\doteq$ to denote syntactic equality), the negation of C is $\neg C \doteq \sum_j a_j \ell_j \leq b - 1$. For a *partial assignment* ρ mapping variables to $\{0, 1\}$, we write $C\restriction_\rho$ for the *restricted constraint* obtained by replacing variables in C assigned by ρ by their values and simplifying the result. We say that C *unit propagates* ℓ *under* ρ if $C\restriction_\rho$ cannot be satisfied unless ℓ is assigned to 1. If unit propagation on all constraints in $\mathcal{C} \cup \mathcal{D} \cup \{\neg C\}$ starting with the empty assignment $\rho = \emptyset$, and extending ρ with new assignments as long as new literals propagate, leads to contradiction in the form of a violated constraint, then we say that C follows by *reverse unit propagation (RUP)* from $\mathcal{C} \cup \mathcal{D}$. Such (efficiently verifiable) RUP steps are allowed in VeriPB proofs when it is convenient to avoid writing out an explicit derivation of C from $\mathcal{C} \cup \mathcal{D}$. We will also write $C\restriction_\omega$ to denote the result of applying to C a *(partial) substitution* ω which can remap variables to other literals in addition to 0 and 1, and we extend this notation to sets in the obvious way by taking unions.

In addition to the cutting planes rules, which can only derive semantically implied constraints, VeriPB has a *redundance-based strengthening rule* that can derive a non-implied constraint C as long as this does not change the feasibility or optimal value of the problem. Formally, C can be derived from $\mathcal{C} \cup \mathcal{D}$ using this rule by exhibiting in the proof a *witness substitution* ω together with subproofs

$$\mathcal{C} \cup \mathcal{D} \cup \{\neg C\} \vdash (\mathcal{C} \cup \mathcal{D} \cup \{C\})\restriction_\omega \cup \{f \geq f\restriction_\omega\}, \tag{1}$$

of all constraints on the right-hand side from the premises on the left-hand side using the derivation rules above. Intuitively, what (1) shows is that if α is any assignment that satisfies $\mathcal{C} \cup \mathcal{D}$ but violates C, then $\alpha \circ \omega$ satisfies $\mathcal{C} \cup \mathcal{D} \cup \{C\}$ and yields at least as good a value for the objective function f.

During presolving, constraints in the input formula can be deleted or replaced by other constraints, and the proof needs to establish that such modifications are correct. While deletions from the derived set $\mathcal{D}$ are always in order, removing a constraint from the core set $\mathcal{C}$ could potentially introduce spurious solutions. Therefore, deleting a constraint C from $\mathcal{C}$ can only be done by the *checked deletion rule*, which requires to show that C could be rederived from $\mathcal{C} \setminus \{C\}$ by redundance-based strengthening (see [BGMN23] for a more detailed explanation).

2.2 A New Rule for Objective Function Updates

When variables are fixed or identified during the presolving process, the objective function f can be modified to a function f'. This modified objective f' can then be used in other presolver reasoning. This scenario arises also in, e.g., MaxSAT solving, and can be dealt with by deriving two PB constraints $f \geq f'$ and $f' \geq f$ in the proof, which encodes that the old and new objective are equal [BBN+23]. Whenever the solver argues in terms of f', a telescoping-sum argument with $f' = f$ can be used to justify the same conclusion in terms of the old objective.

However, if the presolver changes f to f' and then uses reasoning that needs to be certified by redundance-based strengthening, then tricky problems can

Table 1: *Examples of basic derivation rules in VeriPB syntax. Here, (id) refers to the constraint ID assigned by VeriPB.*

Rule	Syntax	Explanation
cutting planes in reverse Polish notation	`pol x1 4 +`	add $x_1 \geq 0$ and (4)
	`pol 3 2 d`	divides (3) by 2
	`pol 1 2 * ~x1 +`	multiplies (1) by 2 and adds $\overline{x}_1 \geq 0$
redundance-based strengthening	`red +1 x1 >= 1; x1 1`	verifies $x_1 \geq 1$ with $\omega = \{x_1 \mapsto 1\}$
	`red +1 x1 +1 x2 >= 1; x1 x2 x2 x1`	verifies $x_1 + x_2 \geq 1$ with $\omega = \{x_1 \mapsto x_2, x_2 \mapsto x_1\}$
RUP	`rup +1 x1 +1 x2 >= 1;`	verifies $x_1 + x_2 \geq 1$ with RUP
move to core	`core id 3`	moves (3) to the core constraints
deletion from core	`delc 3`	deletes (3) from the core constraints
objective function update	`obju new +1 x1 +1 x2 1;`	defines $x_1 + x_2 + 1$ as new objective
	`obju diff +1 ~x1;`	adds $\overline{x}_1$ to the objective

arise. One of the required proof goals in (1) is that the witness ω cannot worsen the objective. If ω does not mention variables in f', then this is obvious to the presolver—ω has no effect on the objective—but if ω assigns variables in the original objective f, then one still needs to derive $f \geq f\!\upharpoonright_\omega$ in the formal proof, which can be challenging. While this can often be done by enlarging the witness ω to include earlier variable fixings and identifications, the extra bookkeeping required for this quickly becomes a major headache, and results in the proof deviating further and further from the actual presolver reasoning that the proof logging is meant to certify.

For this reason, a better solution is to introduce a new *objective function update rule* that formally replaces f by a new objective f', so that all future reasoning about the objective can focus on f' and ignore f. Such a rule needs to be designed with care, so that the optimal value of the problem is preserved. Due to space constraints we cannot provide a formal proof here, but recall that intuitively we maintain the invariant for the core set C that it has the same optimal value as the original problem. In agreement with this, the formal requirement for updating the objective from f to f' is to present in the proof log derivations of the two constraints $f \geq f'$ and $f' \geq f$ from the core set C only.

3 Certifying Presolve Reductions

We now describe how feasibility- and optimality-based presolving reductions can be certified by using VeriPB proof logging enhanced with the new objective function update rule described in Sec. 2.2 above. We distinguish between *primal* and *dual* reductions, where primal reductions strengthen the problem formulation by tightening the convex hull of the problem and preserve all feasible solutions, and dual reductions may additionally remove feasible solutions using optimality-based arguments. More precisely, *weak* dual reductions preserve all optimal solutions,

but may remove suboptimal solutions. *Strong* dual reductions may remove also optimal solutions as long as at least one optimal solution is preserved in the reduced problem. Our selection of methods is motivated by the recent MIP solver implementation described in [PaP]. Before explaining the individual presolving techniques and their certification, we introduce a few general techniques that are needed for the certification of several presolving methods.

3.1 General Techniques

Substitution. In order to reduce the number of variables, constraints, and non-zero coefficients in the constraints, many presolving techniques first try to identify an equality $E \doteq x_k = \sum_{j \neq k} \alpha_j x_j + \beta$ with $\alpha_j, \beta \in \mathbb{Q}$. Subsequently, all occurrences of x_k in the objective and constraints besides E are substituted by the affine expression on the right-hand side and x_k is removed from the problem. The simplest case when x_k is fixed to zero or one, i.e., when $\beta \in \{0, 1\}$ and all $\alpha_j = 0$, is straightforward to handle by deriving a new lower or upper bound on x_k. During presolving, every fixed variable is removed from the model. In the cases where some $\alpha_j \neq 0$, first the equation is expressed as a pair of constraints $E_{\geq} \wedge E_{\leq}$ and then the variable is removed by aggregation as follows.

Aggregation. In order to substitute variables or reduce the number of non-zero coefficients, certain presolving techniques add a scaled equality $s \cdot E \doteq s \cdot E_{\geq} \wedge s \cdot E_{\leq}$, $s \in \mathbb{Q}$, to a given constraint D. We call this an *aggregation*. Since VeriPB certificates expect inequalities with integer coefficients, s is split into two integer scaling factors $s_E, s_D \in \mathbb{Z}$ with $s = s_E/s_D$. In the certificate, the aggregation is expressed as a newly derived constraint

$$D_{new} \doteq \begin{cases} |s_E| \cdot E_{\geq} + |s_D| \cdot D & \text{if } \frac{s_D}{s_E} > 0 \\ |s_E| \cdot E_{\leq} + |s_D| \cdot D & \text{otherwise} \end{cases} .$$

Note that the presolving algorithm may decide to keep working with the constraint $(1/s_D)D_{new}$ internally. In this case, it must store the scaling factor s_D in order to correctly translate between its own state and the state in the certificate; this happens in the implementation used in Sec. 4.

Checked Deletion. The derivation of a new constraint D_{new} can render a previous constraint D redundant. A typical example is the case of substituting a variable above. In a (pre)solver, the previous constraint is overwritten, and in order to keep the constraint database in the proof aligned with the solver, one may want to delete the previous constraint from the proof. In order to check the deletion of D, a subproof is required that proves its redundancy. In most cases, this subproof contains the "inverted" derivation of D_{new}. As an example, consider an aggregation $D_{new} \doteq D + E_{\leq}$ with an equality $E \doteq E_{\leq} \wedge E_{\geq}$. In this case, the subproof for the checked deletion is $D_{new} + E_{\geq}$. Unless stated otherwise, the new constraints are moved to the core and redundant constraints are always removed by inverting the derivation of the constraint that replaces them.

3.2 Primal Reductions

Primal reductions can be certified purely by implicational reasoning.

Bound Strengthening. This preprocessor [FM05, Sav94] tries to tighten the variable domains by iteratively applying well-known *constraint propagation* to all variables in the linear constraints. Each reduced variable domain is communicated to the affected constraints and may trigger further domain changes. This process is continued until no further domain reductions happen or the problem becomes infeasible due to empty domains. Specifically, for an inequality constraint

$$\sum_{j \in N} a_j x_j \geq b \tag{2}$$

with $a_k \neq 0$, we first underestimate $a_k x_k$ via

$$a_k x_k \geq b - \sum_{j \neq k} a_j x_j \geq b - \sum_{j \neq k, a_j > 0} a_j \,.$$

If $a_k > 0$, this yields the lower bound

$$x_k \geq \left\lceil \left(b - \sum_{j \neq k, a_j > 0} a_j\right) / a_k \right\rceil, \tag{3}$$

and if $a_k < 0$ we can obtain an analogous upper bound on x_k.

The bound change can be proven either by RUP, or more explicitly by stating the additions and division needed to form (3) from (2) and the bound constraints. We analyze the effect of both variants in Sec. 4.4.

Parallel Rows. Two constraints C_j and C_k are parallel if a scalar $\lambda \in \mathbb{R}^+$ exists with $\lambda(a_{j1}, \ldots, a_{jn}, b_j) = (a_{k1}, \ldots, a_{kn}, b_k)$. Hence, one of these constraints is redundant and can be removed from the model [ABG+19, GCW+20]. The subproof for deleting the redundant rows must contain the remaining parallel row and λ to prove the redundancy. For a fractional λ the two constraint are scaled to ensure integer coefficients in the certificate.

Probing. The general idea of *probing* [Ach07, Sav94] is to tentatively fix a variable x_j to 0 or 1 and then apply constraint propagation to the resulting model. Suppose x_k is an arbitrary variable with $k \neq i$, then we can learn fixings or implications in the following cases:

1. If $x_j = 0$ implies $x_k = 1$ and $x_j = 1$ implies $x_k = 0$ we can add the constraint $x_j = 1 - x_k$. Analogously, we can derive $x_k = x_j$ in the case that $x_j = 0$ implies $x_k = 0$ and $x_j = 1$ implies $x_k = 1$.

2. If $x_j = 0$ propagates to infeasibility we can fix $x_j = 1$. Analogously, if $x_j = 1$ propagates to infeasibility we can fix $x_j = 0$.

3. If $x_j = 0$ implies $x_k = 0$ and $x_j = 1$ implies $x_k = 0$ we can fix x_k to 0. Analogously, x_k can be fixed to 1 if $x_j = 0$ implies $x_k = 1$ and $x_j = 1$ implies $x_k = 1$.

Cases 1 and 2 can be proven with RUP. To prove correctness of fixing $x_k = 1$ in Case 3 we first derive two new constraints $x_k + x_j \geq 1$ and $x_k - x_j \geq 0$ in the proof log by RUP. Adding these two constraints leads to $x_k \geq 1$. To prove $x_k = 0$ we derive the constraints $x_k + x_j \leq 0$ and $x_k - x_j \leq 0$ leading to $x_k = 0$.

Simple Probing. On equalities with a special structure, a more simplified version of probing called *simple probing* [ABG⁺19, Sec. 3.6] can be applied. Suppose the equation

$$\sum_{j \in N} a_j x_j = b \text{ with } \sum_{j \in N} a_j = 2 \cdot b \text{ and } |a_k| = \sum_{j \in N, a_j > 0} a_j - b$$

holds for a variable x_k with $a_k \neq 0$. Let $\hat{N} = \{p \in N \mid a_p \neq 0\}$. Under these conditions, $x_k = 1$ implies $x_p = 0$ and $x_k = 0$ implies $x_p = 1$ for all $p \in \hat{N}$ with $a_p > 0$. Further, $x_k = 1$ implies $x_p = 1$ and $x_k = 0$ implies $x_p = 0$ for all $p \in \hat{N}$ with $a_p < 0$. These implications can be expressed by the constraints

$$x_k = 1 - x_p \text{ for all } p \in \hat{N} \text{ with } a_p > 0\,, \tag{4}$$

$$x_k = x_p \text{ for all } p \in \hat{N} \text{ with } a_p < 0\,. \tag{5}$$

The constraints (4) and (5) can be proven with RUP and used to substitute variables x_p for all $p \in \hat{N}$ from the problem.

Sparsifying the Matrix. The presolving technique sparsify [ABG⁺19, CM93] tries to reduce the number of non-zero coefficients by adding (multiples of) equalities to other constraints using aggregations. This can be certified as described in Sec. 3.1.

Coefficient Tightening. The goal of this MIP presolving technique, which goes back to [Sav94], is to tighten the LP relaxation, i.e., the relaxation obtained when the integrality requirements are replaced by $x_j \in [0, 1]$. To this end, the coefficients of constraints are modified such that LP relaxation solutions are removed, but all integer feasible solutions are preserved. Suppose we are given a constraint $\sum_{j \in N} a_j x_j \geq b$ with $a_k \geq \varepsilon := a_k - b + \sum_{j \neq k, a_j < 0} a_j > 0$, then the constraint can be strengthened to $(a_k - \varepsilon)x_k + \sum_{j \neq k} a_j x_j \geq b$. The case $a_k < 0$ is handled analogously. This technique is also known as *saturation* in the SAT community [CK05] and VeriPB provides a dedicated saturation rule that can be used directly for proving the correctness of coefficient tightening. The deletion of the original, weaker constraint can be proven automatically.

GCD-based Simplification. This presolving technique from [Wen16] uses a divisibility argument to first eliminate variables from a constraint and then tighten its right-hand side. Given $C \doteq \sum_{j \in N} a_j x_j \geq b$ with $|a_1| \geq \cdots \geq |a_n| > 0$. We define the greatest common divisor $g_k = \gcd(a_1, \ldots, a_k)$ as the largest value g such that $a_j / g \in \mathbb{Z}$ for all $j \in \{1, \ldots, k\}$. If for an index k it holds that $b - g_k \cdot \left\lceil \frac{b}{g_k} \right\rceil \geq \sum_{k < j \leq n, a_j > 0} a_j$ and $b - g_k \cdot \left\lceil \frac{b}{g_k} \right\rceil - g_k \leq \sum_{k < j \leq n, a_j < 0} a_j$, then all $a_{k+1}, \ldots, a_n$ can be set to 0. This first step can be certified as *weakening* [LBMW20] and VeriPB provides an out-of-the-box verification function for it. Finally, b can be rounded to $g_k \cdot \lceil b / g_k \rceil$.

This rounding step can be certified by dividing C with g_k and then multiply it again with g_k.

Substituting Implied Free Variables. A variable x_j is called *implied free* if its lower bound and its upper bound can be derived from the constraints. For example, the constraints $x_1 - x_2 \geq 0$ and $x_2 \geq 0$ imply the lower bound $x_1 \geq 0$. If we have an implied free variable x_j in an equality $E \doteq a_j x_j + \sum_{k \neq j} a_k x_k = b$ with $a_j > 0$, then we can remove x_j from the problem by substituting it with $x_j = \left(b - \sum_{k \neq j} a_k x_k\right)/a_j$, see [ABG+19] for details.

To apply the substitution in the certificate we use aggregations to remove x_j from all constraints and the objective function update to remove x_j from the objective. If coefficients c_j/a_j or a_k/a_j are non-integer, then the resulting constraints are scaled as described in Sec. 3.1. To prove the deletion of E, we derive two constraints by adding $x_j \geq 0$ and $1 \geq x_j$ to E each, which results in

$$b \geq \sum_{k \neq j} a_k x_k \;\wedge\; \sum_{k \neq j} a_k x_k \geq b - a_j\,. \tag{6}$$

Then the deletion of $E_{\geq}$ can be certified by a witness $\omega = \{x_j \mapsto 1\}$. The constraint simplifies to (6) and is therefore fulfilled. Analogously, we use the witness $\omega = \{x_j \mapsto 0\}$ to certify the deletion of $E_{\leq}$. Finally, to delete the constraints in (6) we generate a subproof that shows that negation of the auxiliary constraints in (6) leads to $x_j \notin \{0, 1\}$. This is a contradiction to the implied variable bounds $0 \leq x_j \leq 1$. Since these bounds are still present through the implying constraints, we can add these implying constraints to (6) in the subproof to arrive at a contradiction.

Singleton Variables. It is well-known that variables that appear only in one inequality constraint or equality can be removed from the problem [ABG+19, Sec. 5.2]. This can be certified by applying one of the following primal or dual strategies in this order: First, try to apply duality-based fixing, see Sec. 3.3; second, an implied free singleton variable can be substituted as explained above; otherwise, the singleton variable can be treated as a *slack variable*: substitute the variable in the objective, then relax the equality as in (6), and delete the original constraint.

3.3 Dual Reductions

Dual reductions remove solutions while preserving at least one optimal solution. Hence, to prove the correctness of dual reductions we need to involve the redundance-based strengthening rule of VeriPB. For each derived constraint C we only explain how to prove $f \geq f\restriction_\omega$ (subject to the negation $\neg C$); the proof goals for $C\restriction_\omega$ can be derived in a very similar fashion.

Duality-based Fixing. This presolving step described in [ABG+19, Sec. 4.2] counts the *down-* and *up-lock* of a variable. A down-lock on variable x_j is a negative coefficient, an up-lock on variable x_j is a positive coefficient (for $\geq$ constraints). If x_j has no down-locks and $c_j \leq 0$, it can be fixed to zero; if x_j has no up-locks and $c_j \geq 0$, it can be fixed to one. These reductions can be certified with redundance-based strengthening using the witness $\omega = \{x_j \mapsto v\}$, where v is the fixing value.

The proof goal for $f \geq f\restriction_\omega$ is equivalent to $c_j x_j \geq c_j v$, which is fulfilled by the conditions of duality-based fixing.

Dominated Variables. A variable x_j is said to *dominate* another variable x_k [ABG+19, GKM+15], in notation $x_j \succ x_k$, if

$$c_j \leq c_k \;\wedge\; a_{ij} \geq a_{ik} \text{ for all } i \in \{1, \ldots, m\}\,, \tag{7}$$

where a_{ij} and a_{ik} are the coefficients of variable x_j and x_k, respectively, in the i-th constraint. Variable x_j is then favored over x_k since x_j contributes less to the objective function, but more to the feasibility of the constraints. For every domination $x_j \succ x_k$, a constraint $C \doteq x_j \geq x_k$ can be introduced. This constraint can be certified by redundance-based strengthening with the witness $\omega = \{x_k \mapsto x_j, x_j \mapsto x_k\}$. The proof goal for $f \geq f\restriction_\omega$ is equivalent to

$$c_j x_j + c_k x_k \geq c_j x_k + c_k x_j\,. \tag{8}$$

The negated constraint $\neg C \doteq x_j < x_k$ leads to $x_k = 1$ and $x_j = 0$. Substituting these values in (8) leads to $c_k \geq c_j$, which follows directly from Condition (7).

Dominated Variables Advanced. For an implied free variable we can drop the variable bounds and pretend the variable is unbounded. This allows for additional fixings in the following cases of dominated variables:

(a) If the upper bound of x_j is implied and $x_j \succ x_k$, then $x_k = 0$.

(b) If the lower bound of x_k is implied and $x_j \succ x_k$, then $x_j = 1$.

(c) If the upper bound of x_j is implied and $x_j \succ -x_k$, then $x_k = 1$.

(d) If the lower bound of x_j is implied and $-x_j \succ x_k$, then $x_j = 0$.

We use redundance-based strengthening with witness $\omega = \{x_k \mapsto 0\}$ to prove the correctness of a as follows. If the upper bound of x_j is implied, this means there exists a constraint with $a_{ij} < 0$ such that $x_j \leq \left\lfloor \frac{b_\ell - \sum_{\ell \neq j, a_{i\ell} > 0} a_{i\ell}}{a_{ij}} \right\rfloor = 1$. Due to Condition (7), it must hold that $0 > a_{ij} \geq a_{ik}$, and the constraint $x_j + x_k \leq 1$ can be derived. Hence, negating and propagating $C \doteq x_k = 0$ with RUP leads to contradiction, which proves the validity of C. Case b can be handled analogously using the witness $\omega = \{x_k \mapsto 1\}$. To derive $C \doteq x_k = 1$ in c we use redundance-based strengthening with witness $\omega = \{x_k \mapsto 1, x_j \mapsto 1\}$. Then, the proof goal for $f \geq f\restriction_\omega$ is $c_j \cdot x_j + c_k \cdot x_k \geq c_j + c_k$. After propagating $\neg C$, this becomes equivalent to $c_j \leq -c_k$, which is true by Condition (7). Case d can be handled analogously using the witness $\omega = \{x_k \mapsto 0, x_j \mapsto 0\}$.

3.4 Example

We conclude this section with an example of a small certificate for the substitution of an implied free variable in Fig. 1, also available with a more detailed description at the software repository of PaPILO [Hoe23]. Consider the 0–1 ILP

$$\min\; x_1 + x_2 \;\text{ s.t. }\; x_1 + x_2 - x_3 - x_4 = 1\,, \tag{9}$$

$$-x_1 + x_5 \geq 0\,, \tag{10}$$

```
* generates ID 4:
pol 1 ~x1 + ;
core id 4
* generates ID 5:
pol 2 x1 + ;
core id 5
```

```
* generates ID 6:
pol 3 1 + ;
core id 6
delc 3 ;  ; begin
   pol 6 2 +
end
obju new +1 x3 +1 x4 1 ;
```

```
delc 2 ; x1 -> 0
delc 1 ; x1 -> 1
delc 5
delc 4 ; ; begin
   pol 6 -1 +
end
```

Figure 1: *A VeriPB certificate to substitute an implied free variable x_1.*

in which the lower bound of x_1 is implied by (9) and the upper bound of x_1 is implied by (10). Hence, x_1 is implied free and we can use (9) to substitute it.

In the left section of Fig. 1 we first derive the two auxiliary constraints

$$0 \le x_2 - x_3 - x_4 \le 1\,, \tag{11}$$

which receives the constraint IDs 4 and 5 and are moved to the core. Note that the equality in (9) is split into two inequalities with IDs 1 and 2. In the middle section, we first remove x_1 from (10) by aggregation with (9), perform checked deletion, then remove x_1 from the objective (automatically proven by VeriPB). Last, in the right section, we delete the equality in (9) used for the substitution and the auxiliary constraints in (11) and arrive at the reformulated problem min $x_3 + x_4 + 1$ s.t. $x_2 - x_3 - x_4 + x_5 \ge 1$. From here, we could continue to derive $x_2 = 1$ by duality-based fixing, since x_2 has zero up-locks and objective coefficient zero. This displays the importance of the objective update, as without it x_2 would still contribute to the objective with a positive coefficient, and this would prohibit duality-based fixing to 1.

4 Computational Study

In this section we quantify the cost of *certifying* presolve reductions in a state-of-the-art implementation for MIP-based presolve (Sec. 4.2) and the cost of *verifying* the resulting certificates (Sec. 4.3). In Sec. 4.4, we analyze the impact of certifying constraint propagation by RUP or by an explicit cutting planes proof.

4.1 Experimental Setup

For generating the presolve certificates we use the solver-independent presolve library PaPILO [PaP], which provides a large set of MIP and LP techniques from the literature, described in Sec. 3. Additionally, it accelerates the search for presolving reductions by parallelization, encapsulating each reduction in a so-called transaction to avoid expensive synchronization [GGH23]. Logging the certificate, however, is performed sequentially while evaluating the transactions.

We base our experiments on models from the Pseudo-Boolean Competition 2016 [Rou16] including 1398 linear small integer decision and 532 linear small integer optimization instances of the competitions PB10, PB11, PB12, PB15, and PB16 and 295 decision and 145 optimization instances from MIPLIB 2017 [GHG+21]

in the OPB translation [Dev20], excluding 10 large-scale instances[1] for which PaPILO reaches the memory limit. This yields a total of 671 optimization and 1681 decision instances. We use PaPILO 2.2.0 [HG23] running on 6 threads and VeriPB 2.0 [GO23]. The experiments are carried out on identical machines with an 11th Gen Intel(R) Core(TM) i5-1145G7 @ 2.60 GHz CPU and 16 GB of memory and are assigned 14,000 MB of memory. The strict time limit for presolve plus certification and verification is three hours. Times (reported in seconds) do not include the time for reading the instance file. For all aggregations, we use the shifted geometric mean with a shift of 1 second.

4.2 Overhead of Proof Logging

In the first experiment, we analyze the overhead of proof logging in PaPILO. The average results are summarized in Tab. 2, separately over decision (dec) and optimization (opt) instances for PB16 and MIPLIB. Column "relative" indicates the average slow-down incurred by printing the certificate.

The relative overhead of proof logging is less than 6% across all test sets. VeriPB supports two variants to change the objective function. Either printing the entire objective (`obju new`) or printing only the changes in the objective (`obju diff`). In our experiments, we only print the changes, since printing the entire objective for each change can lead to a large certificate and overhead, especially for instances with large and dense objective functions. On the PB16 instance normalized-datt256, for example, PaPILO finds 135 206 variable fixings. Updating the entire objective function with 262 144 non-zeros for each of these variables leads to a huge certificate of about 138 GB and increases the time from 3.3 seconds (when printing only the changes) to 6625 seconds.[2]

For 99% of the instances, we can further observe that the *overhead per applied reduction* is below $0.001 \cdot 10^{-3}$ seconds over both test sets. This means that the proof logging overhead is not only small on average, but also small per applied reduction on the vast majority of instances. These results show that the overhead scales well with the number of applied reductions and that proof logging remains viable even for instances with many transactions. Here, under applied reductions we subsume all applied transactions and each variable fixing or row deletion in the first model clean-up phase. During model clean-up, PaPILO fixes variables and removes redundant constraints from the problem. While PaPILO technically does not count these reductions as full transactions found during the parallel presolve phase, their certification can incur the same overhead.

4.3 Verification Performance on Presolve Certificates

In this section, we analyze the time to verify the certificates generated by PaPILO. The results are summarized in Tab. 3. The "verified" column lists the number of instances verified within 3 hours. VeriPB timeouts are counted as twice the time

[1] normalized-184, normalized-pb-simp-nonunif, a2864-99blp, ivu06-big, ivu59, supportcase11, a2864-99blp.0.s/u, supportcase11.0.s/u

[2] Certificate generated on Intel Xeon Gold 5122 @ 3.60GHz 96 GB with 50,000 MB of memory assigned.

Table 2: *Runtime comparison of PaPILO with and without proof logging.*

test set	size	default [s]	w/proof log [s]	relative
PB16-dec	1397	0.06	0.06	1.00
MIPLIB-dec	291	0.42	0.43	1.02
PB16-opt	531	0.65	0.66	1.02
MIPLIB-opt	142	0.33	0.35	1.06

Table 3: *Time to verify the certificates. VeriPB timeouts are treated with PAR2.*

			PaPILO time [s]		VeriPB	relative time w.r.t.	
test set	size	verified	default	w/proof log	time [s]	default	w/proof log
PB16-dec	1397	1397	0.06	0.06	0.88	14.67	14.67
MIPLIB-dec	291	267	0.42	0.43	9.64	22.85	22.42
PB16-opt	531	520	0.65	0.66	10.44	16.06	15.82
MIPLIB-opt	142	139	0.33	0.35	5.25	15.91	15.00

limit, i.e., PAR2 score. Similar to Tab. 2, the "relative" columns report the relative overhead of VeriPB runtime compared to PaPILO.

First note that all certificates are verified by VeriPB (partially on the 38 instances where VeriPB times out). On average, it takes between 14.7 and 22.4 times as much time to verify the certificates than to produce them. Nevertheless, some instances take a longer than average time to verify. Over all test sets, 25% of the instances have an overhead of at least a factor of 193, see also Fig. 2.

To put this result into context, note that presolving amounts more to a transformation than to a (partial) solution of the problem. Each reduction has to be certified and verified while a purely solution-targeted algorithm may be able to skip certifying of a larger part of the findings that are not form a part of the final proof of optimality. Hence, it makes sense to compare the performance of VeriPB on presolve certificates to the overhead for, e.g., for verifying CNF translations [GMNO22]. For this study, a similar performance overhead is reported as in Fig. 2.

4.4 Performance Analysis on Constraint Propagation

Finally, we investigate how the performance of VeriPB depends on whether we use RUP (as in Sec. 4.2 and Sec. 4.3) or explicit cutting planes derivations (POL) to certify bound strengthening reductions from constraint propagation. Here, we additionally exclude 9 large-scale instances[3] for which PaPILO reaches the memory limit when certifying with POL. The results are summarized in Tab. 4. The "verified" column contains the number of instances verified by VeriPB within the time limit. The "time" column reports the time for verification.

Deriving the propagation directly with cutting planes is 3.2% faster on PB16-dec, 2.8% faster on MIPLIB-dec, 13.1% faster on MIPLIB-opt, and 0.7% faster on

[3] neos-4754521-awarau.0.s, neos-827015.0.s/u, neos-829552.0.s/u, s100.0.s/u, normalized-datt256, s100

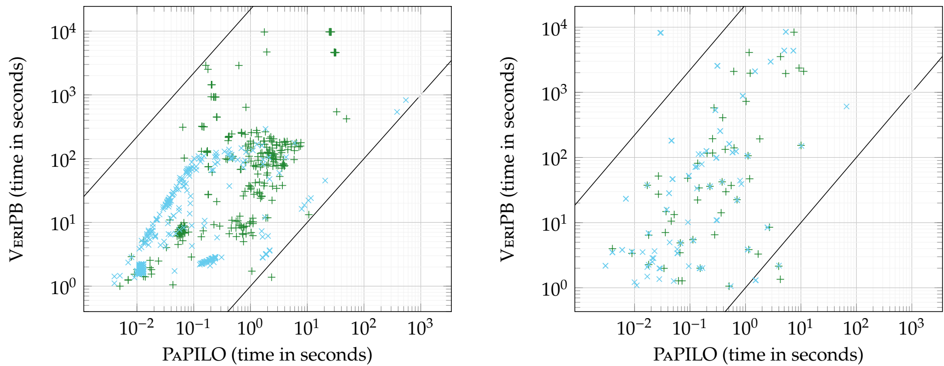


Figure 2: *Running times of VeriPB vs. PaPILO on test sets PB16 (left) and MIPLIB (right), including all instances with more than 1 seconds in VeriPB and less than 30 minutes in PaPILO, and excluding timeouts. Green + signs mark optimization and blue × signs mark decision instances.*

Table 4: *Comparison of the runtime of VeriPB with RUP and POL over instances with at least 10 propagations.*

		RUP		POL		
test set	size	verified	time [s]	verified	time [s]	relative
PB16-dec	284	284	2.21	284	2.14	0.968
MIPLIB-dec	35	31	153.23	31	148.88	0.972
PB16-opt	153	142	28.43	142	28.22	0.993
MIPLIB-opt	16	14	147.11	14	127.83	0.869

PB16-opt. On 95% of the decision instances using RUP is at most 9.7% slower. While it is expected that verification is faster when the cutting planes proof is given explicitly, it is surprising that the performance difference between the methods is not more pronounced.This is partly due to the cost of the watched-literal scheme [MMZ+01, SS06] used by VeriPB for unit propagation. The overhead of maintaining the watches is present regardless of whether (reverse) unit propagation is used or not. Furthermore, unit propagation is also used for automatically verifying redundance-based strengthening. Together, this limits the potential for runtime savings by providing the explicit cutting planes proof.

Furthermore, providing an explicit cutting planes proof for propagation requires printing the constraint into the certificate. Hence, the certificate size becomes dependent on the number of non-zeros in the constraints leading to propagations. In contrast, the overhead of RUP is constant and much smaller.

All in all, these results suggest to prefer RUP when deriving constraint propagation since it barely impacts the performance of VeriPB and keeps the size of the certificate smaller. The computational cost of RUP could be further reduced by extending it to accept an ordered list of constraints that shall be propagated first, similar as in [CFHH+17]. Such an extension could also be used for other presolving techniques, in particular probing and simple probing.

5 Conclusion

In this paper we set out to demonstrate how presolve techniques from state-of-the-art MIP solvers can be equipped with certificates in order to verify the equivalence between original and reduced models. Although the pseudo-Boolean proof logging format behind VeriPB [BGMN22] was not designed with this purpose in mind, we could show that a limited extension needed for handling updates of the objective function is sufficient to craft a certified presolver for 0–1 ILPs.

However, our experimental study on instances from pseudo-Boolean competitions and MIPLIB also exhibited that the verification of MIP-based presolving can suffer from large and overly verbose certificates. To shrink the proof size we introduced a sparse objective update function but identified further possible improvements. First, a native substitution rule in VeriPB would remove the need for the explicit derivation of new aggregations and the verification of checked deletion as described in Sec. 3.1. For instances where presolving is dominated by substitutions, we estimate that this would reduce certificate sizes by up to 90%, and no more time would be spent on checked deletion for substitutions. Second, augmenting the RUP syntax by the option to specify an ordered list of constraints to propagate first, similarly as in [CFHH+17], would accelerate RUP, in particular for fast verification of bound strengthenings by constraint propagation.

While VeriPB is currently restricted to operate on integer coefficients only, the certification techniques presented in Sec. 3 do not rely on this assumption and are applicable to general binary programs. It has been shown how to construct VeriPB certificates for bounded integer domains [GMN22, MM23], and within the framework of the generalized proof system laid out in [DEGH23], our certificates would even translate to continuous and unbounded integer domains. To conclude, we believe our results show convincingly that this type of proof logging techniques is a very promising direction of research also for MIP presolve beyond 0–1 ILPs.

Acknowledgements

The authors wish to acknowledge helpful technical discussions on VeriPB in general and the objective update rule in particular with Bart Bogaerts, Ciaran McCreesh, and Yong Kiam Tan. The work for this article has been partly conducted within the Research Campus MODAL funded by the German Federal Ministry of Education and Research (BMBF grant number 05M14ZAM). Jakob Nordström was supported by the Swedish Research Council grant 2016-00782 and the Independent Research Fund Denmark grant 9040-00389B. Andy Oertel was supported by the Wallenberg AI, Autonomous Systems and Software Program (WASP) funded by the Knut and Alice Wallenberg Foundation. The computational experiments were enabled by resources provided by LUNARC at Lund University.

References

[ABG+19] Tobias Achterberg, Robert Bixby, Zonghao Gu, Edward Rothberg, and Dieter Weninger. Presolve reductions in mixed integer programming. *INFORMS Journal on Computing*, 32, 11 2019.

[Ach07] Tobias Achterberg. *Constraint Integer Programming*. Doctoral thesis, Technische Universität Berlin, Fakultät II - Mathematik und Naturwissenschaften, Berlin, 2007.

[AGJ+18] Özgür Akgün, Ian P. Gent, Christopher Jefferson, Ian Miguel, and Peter Nightingale. Metamorphic testing of constraint solvers. In *Proceedings of the 24th International Conference on Principles and Practice of Constraint Programming (CP '18)*, volume 11008 of *Lecture Notes in Computer Science*, pages 727–736. Springer, August 2018.

[BBN+23] Jeremias Berg, Bart Bogaerts, Jakob Nordström, Andy Oertel, and Dieter Vandesande. Certified core-guided MaxSAT solving. In *Proceedings of the 29th International Conference on Automated Deduction (CADE-29)*, volume 14132 of *Lecture Notes in Computer Science*, pages 1–22. Springer, July 2023.

[BGMN22] Bart Bogaerts, Stephan Gocht, Ciaran McCreesh, and Jakob Nordström. Certified symmetry and dominance breaking for combinatorial optimisation. In *Proceedings of the 36th AAAI Conference on Artificial Intelligence (AAAI '22)*, pages 3698–3707, February 2022.

[BGMN23] Bart Bogaerts, Stephan Gocht, Ciaran McCreesh, and Jakob Nordström. Certified dominance and symmetry breaking for combinatorial optimisation. *Journal of Artificial Intelligence Research*, 77:1539–1589, August 2023. Preliminary version in *AAAI '22*.

[BHvMW21] Armin Biere, Marijn J. H. Heule, Hans van Maaren, and Toby Walsh, editors. *Handbook of Satisfiability*, volume 336 of *Frontiers in Artificial Intelligence and Applications*. IOS Press, 2nd edition, February 2021.

[BJK21] Armin Biere, Matti Järvisalo, and Benjamin Kiesl. Preprocessing in SAT solving. In Biere et al. [BHvMW21], chapter 9, pages 391–435.

[BN21] Samuel R. Buss and Jakob Nordström. Proof complexity and SAT solving. In Biere et al. [BHvMW21], chapter 7, pages 233–350.

[CCT87] William Cook, Collette Rene Coullard, and György Turán. On the complexity of cutting-plane proofs. *Discrete Applied Mathematics*, 18(1):25–38, November 1987.

[CFHH+17] Luís Cruz-Filipe, Marijn J. H. Heule, Warren A. Hunt, Matt Kaufmann, and Peter Schneider-Kamp. Efficient certified rat verification. In Leonardo de Moura, editor, *Automated Deduction – CADE 26*, pages 220–236, Cham, 2017. Springer International Publishing.

[CGS17] Kevin K. H. Cheung, Ambros M. Gleixner, and Daniel E. Steffy. Verifying integer programming results. In *Proceedings of the 19th International Conference on Integer Programming and Combinatorial Optimization (IPCO '17)*, volume 10328 of *Lecture Notes in Computer Science*, pages 148–160. Springer, June 2017.

[CHH+17] Luís Cruz-Filipe, Marijn J. H. Heule, Warren A. Hunt Jr., Matt Kaufmann, and Peter Schneider-Kamp. Efficient certified RAT verification. In *Proceedings of the 26th International Conference on Automated Deduction (CADE-26)*, volume 10395 of *Lecture Notes in Computer Science*, pages 220–236. Springer, August 2017.

[CK05] Donald Chai and Andreas Kuehlmann. A fast pseudo-Boolean constraint solver. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 24(3):305–317, March 2005. Preliminary version in *DAC '03*.

[CKSW13a] William Cook, Thorsten Koch, Daniel E. Steffy, and Kati Wolter. A hybrid branch-and-bound approach for exact rational mixed-integer programming. *Mathematical Programming Computation*, 5(3):305–344, September 2013.

[CKSW13b] William Cook, Thorsten Koch, Daniel E. Steffy, and Kati Wolter. A hybrid branch-and-bound approach for exact rational mixed-integer programming. *Mathematical Programming Computation*, 5(3):305–344, 2013.

[CM93] S. Frank Chang and S. Thomas McCormick. Implementation and computational results for the hierarchical algorithm for making sparse matrices sparser. *ACM Trans. Math. Softw.*, 19(3):419–441, sep 1993.

[CMS17] Luís Cruz-Filipe, João P. Marques-Silva, and Peter Schneider-Kamp. Efficient certified resolution proof checking. In *Proceedings of the 23rd International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS '17)*, volume 10205 of *Lecture Notes in Computer Science*, pages 118–135. Springer, April 2017.

[DEGH23] Jasper van Doornmalen, Leon Eifler, Ambros Gleixner, and Christopher Hojny. A proof system for certifying symmetry and optimality reasoning in integer programming. Technical Report 2311.03877, arXiv.org, November 2023.

[Dev20] Jo Devriendt. Miplib 0-1 instances in opb format. 05 2020.

[EG22] Leon Eifler and Ambros Gleixner. A computational status update for exact rational mixed integer programming. *Mathematical Programming*, 2022.

[EG24] Leon Eifler and Ambros Gleixner. Safe and verified gomory mixed integer cuts in a rational MIP framework. *SIAM Journal on Optimization*, 34(1):742–763, 2024.

[EGP22] Leon Eifler, Ambros Gleixner, and Jonad Pulaj. A safe computational framework for integer programming applied to chvátal's conjecture. *ACM Transactions on Mathematical Software*, 48(2), 2022.

[FM05] Armin Fügenschuh and Alexander Martin. Computational integer programming and cutting planes. In K. Aardal, G.L. Nemhauser, and R. Weismantel, editors, *Discrete Optimization*, volume 12 of *Handbooks in Operations Research and Management Science*, pages 69–121. Elsevier, 2005.

[GCW+20] Patrick Gemander, Wei-Kun Chen, Dieter Weninger, Leona Gottwald, and Ambros Gleixner. Two-row and two-column mixed-integer presolve using hashing-based pairing methods. *EURO Journal on Computational Optimization*, 8(3-4):205 – 240, 2020.

[GGH23] Ambros Gleixner, Leona Gottwald, and Alexander Hoen. PaPILO: A parallel presolving library for integer and linear programming with multiprecision support. *INFORMS Journal on Computing*, 2023.

[GHG+21] Ambros Gleixner, Gregor Hendel, Gerald Gamrath, Tobias Achterberg, Michael Bastubbe, Timo Berthold, Philipp M. Christophel, Kati Jarck, Thorsten Koch, Jeff Linderoth, Marco Lübbecke, Hans D. Mittelmann, Derya Ozyurt, Ted K. Ralphs, Domenico Salvagnin, and Yuji Shinano. MIPLIB 2017: Data-Driven Compilation of the 6th Mixed-Integer Programming Library. *Mathematical Programming Computation*, 13:443–490, 2021.

[GKM+15] Gerald Gamrath, Thorsten Koch, Alexander Martin, Matthias Miltenberger, and Dieter Weninger. Progress in presolving for mixed integer programming. *Mathematical Programming Computation*, 7, 06 2015.

[GMM+20] Stephan Gocht, Ross McBride, Ciaran McCreesh, Jakob Nordström, Patrick Prosser, and James Trimble. Certifying solvers for clique and maximum common (connected) subgraph problems. In *Proceedings of the 26th International Conference on Principles and Practice of Constraint Programming (CP '20)*, volume 12333 of *Lecture Notes in Computer Science*, pages 338–357. Springer, September 2020.

[GMN20] Stephan Gocht, Ciaran McCreesh, and Jakob Nordström. Subgraph isomorphism meets cutting planes: Solving with certified solutions. In *Proceedings of the 29th International Joint Conference on Artificial Intelligence (IJCAI '20)*, pages 1134–1140, July 2020.

[GMN22] Stephan Gocht, Ciaran McCreesh, and Jakob Nordström. An auditable constraint programming solver. In *Proceedings of the 28th International Conference on Principles and Practice of Constraint Programming (CP '22)*, volume 235 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 25:1–25:18, August 2022.

[GMNO22] Stephan Gocht, Ruben Martins, Jakob Nordström, and Andy Oertel. Certified CNF translations for pseudo-Boolean solving. In *Proceedings of the 25th International Conference on Theory and Applications of Satisfiability Testing (SAT '22)*, volume 236 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 16:1–16:25, August 2022.

[GN21] Stephan Gocht and Jakob Nordström. Certifying parity reasoning efficiently using pseudo-Boolean proofs. In *Proceedings of the 35th AAAI Conference on Artificial Intelligence (AAAI '21)*, pages 3768–3777, February 2021.

[GO23] Stefan Gocht and Andy Oertel. Veripb, 2023. githash: dd7aa5a1.

[GSD19] Xavier Gillard, Pierre Schaus, and Yves Deville. SolverCheck: Declarative testing of constraints. In *Proceedings of the 25th International Conference on Principles and Practice of Constraint Programming (CP '19)*, volume 11802 of *Lecture Notes in Computer Science*, pages 565–582. Springer, October 2019.

[HG23] Alexander Hoen and Leona Gottwald. Papilo: Parallel presolve integer and linear optimization, 2023. githash: 3b082d4.

[HHW13a] Marijn J. H. Heule, Warren A. Hunt Jr., and Nathan Wetzler. Trimming while checking clausal proofs. In *Proceedings of the 13th International Conference on Formal Methods in Computer-Aided Design (FMCAD '13)*, pages 181–188, October 2013.

[HHW13b] Marijn J. H. Heule, Warren A. Hunt Jr., and Nathan Wetzler. Verifying refutations with extended resolution. In *Proceedings of the 24th International Conference on Automated Deduction (CADE-24)*, volume 7898 of *Lecture Notes in Computer Science*, pages 345–359. Springer, June 2013.

[Hoe23] Alexander Hoen. Papilo: Parallel presolve integer and linear optimization, 2023. githash: 5df3dd6d.

[Klo14] Ed Klotz. Identification, assessment, and correction of ill-conditioning and numerical instability in linear and integer programs. In Alexandra Newman and Janny Leung, editors, *Bridging Data and Decisions*, TutORials in Operations Research, pages 54–108. 2014.

[LBMW20] Daniel Le Berre, Pierre Marquis, and Romain Wallon. On weakening strategies for pb solvers. In Luca Pulina and Martina Seidl, editors, *Theory and Applications of Satisfiability Testing – SAT 2020*, pages 322–331, Cham, 2020. Springer International Publishing.

[MM23] Matthew McIlree and Ciaran McCreesh. Proof logging for smart extensional constraints. In *Proceedings of the 29th International Conference on Principles and Practice of Constraint Programming (CP '23)*, volume 280 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 26:1–26:17, August 2023.

[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 Design Automation Conference (DAC '01)*, pages 530–535, June 2001.

[PaP] PaPILO — parallel presolve for integer and linear optimization. `https://github.com/lgottwald/PaPILO`.

[Rou16] Olivier Roussel. Pseudo-boolean competition 2016, 2016.

[Sav94] Martin Savelsbergh. Preprocessing and probing techniques for mixed integer programming problems. *ORSA Journal on Computing*, 6, 11 1994.

[SBD19] Youcef Sahraoui, Pascale Bendotti, and Claudia D'Ambrosio. Real-world hydro-power unit-commitment: Dealing with numerical errors and feasibility issues. *Energy*, 184:91–104, 2019. Shaping research in gas-, heat- and electric- energy infrastructures.

[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, March 2006. Preliminary version in *DATE '05*.

[Ste11] Daniel E. Steffy. *Topics in exact precision mathematical programming*. PhD thesis, Georgia Institute of Technology, 2011.

[VDB22] Dieter Vandesande, Wolf De Wulf, and Bart Bogaerts. QMaxSATpb: A certified MaxSAT solver. In *Proceedings of the 16th International Conference on Logic Programming and Non-monotonic Reasoning (LPNMR '22)*, volume 13416 of *Lecture Notes in Computer Science*, pages 429–442. Springer, September 2022.

[Wen16] Dieter Weninger. *Solving mixed-integer programs arising in production planning*. Phd thesis, Friedrich-Alexander-Universität Erlangen-Nürnberg, 2016.

[WHH14] Nathan Wetzler, Marijn J. H. Heule, and Warren A. Hunt Jr. DRAT-trim: Efficient checking and trimming using expressive clausal proofs. In *Proceedings of the 17th International Conference on Theory and Applications of Satisfiability Testing (SAT '14)*, volume 8561 of *Lecture Notes in Computer Science*, pages 422–429. Springer, July 2014.

End-to-End Verification for Subgraph Solving

Abstract

Modern subgraph-finding algorithm implementations consist of thousands of lines of highly optimized code, and this complexity raises questions about their trustworthiness. Recently, some state-of-the-art subgraph solvers have been enhanced to output machine-verifiable proofs that their results are correct. While this significantly improves reliability, it is not a fully satisfactory solution, since end-users have to trust both the proof checking algorithms and the translation of the high-level graph problem into a low-level 0–1 integer linear program (ILP) used for the proofs.

In this work, we present the first *formally verified* toolchain capable of full end-to-end verification for subgraph solving, which closes both of these trust gaps. We have built encoder frontends for various graph problems together with a 0–1 ILP (a.k.a. pseudo-Boolean) proof checker, all implemented and formally verified in the CakeML ecosystem. This toolchain is flexible and extensible, and we use it to build verified proof checkers for both decision and optimization graph problems, namely, *subgraph isomorphism*, *maximum clique*, and *maximum common (connected) induced subgraph*. Our experimental evaluation shows that end-to-end formal verification is now feasible for a wide range of hard graph problems.

Paper V

1 Introduction

Combinatorial optimization algorithms have improved immensely since the turn of the millennium, and are now routinely used to solve large-scale real-world problems, through both general-purpose solving paradigms [BHvMW21, BR07, GSVW14] and dedicated algorithms for more specialised problems such as subgraph finding [MPT20]. Since these combinatorial solvers are used for an increasingly wide range of applications, it becomes crucial that the results they compute can be trusted. Sadly, this is currently not the case [CKSW13, AGJ⁺18, GSD19, BMN22]. Extensive testing, though beneficial, has not been able to resolve the

Stephan Gocht, Ciaran McCreesh, Magnus O. Myreen, Jakob Nordström, Andy Oertel, and Yong Kiam Tan. "End-to-End Verification for Subgraph Solving". In *Proceedings of the 38th AAAI Conference on Artificial Intelligence (AAAI '24)*, pages 8038–8047, Febuary 2024.

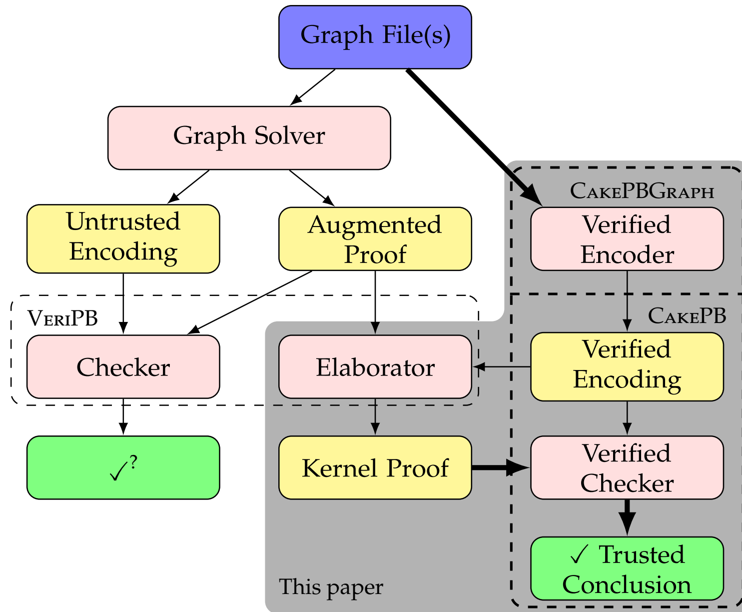


Figure 1: *The full verification workflow. Without verified proof checking, only the left-hand part of the diagram is used. Our current work enables the additional shaded parts, where the thick dashed box is the formally verified program and thick arrows show its key input-output interfaces.*

problem of solvers occasionally producing faulty answers, and attempts to build correct-by-construction software using formal verification run into the obstacle that current techniques cannot scale to the level of complexity of modern solvers.

Instead, the most promising way to achieve verifiably correct combinatorial solving seems to be *proof logging*, meaning that solvers produce efficiently verifiable certificates of correctness that can be corroborated by an independent proof checking program [MMNS11]. This approach has been successfully used in the SAT community [HHW13a, HHW13b, WHH14], which raises the question of whether similar techniques could be employed in other settings such as subgraph finding. For this it would seem that the proof checker would need to understand graph concepts such as vertices, edges, neighbourhoods, et cetera. Surprisingly, this turns out not to be the case—instead, the solver can encode the graph problem using 0–1 linear inequalities (also referred to as *pseudo-Boolean constraints*), and then justify its complex high-level reasoning in terms of this low-level representation. This approach has been used to add proof logging with the VeriPB tool to state-of-the-art solvers for subgraph isomorphism, clique, and maximum common (connected) induced subgraph [GMN20, GMM+20], as illustrated in the left-hand part of Figure 1. We emphasize that although this approach uses reasoning with pseudo-Boolean constraints for the proof logging, it is *not* limited to pseudo-Boolean solving. Rather, it can be used to certify the output of *any* untrusted

solver—such as tools that operate natively on graph representations—as long as the solver's relevant reasoning steps can be expressed with pseudo-Boolean proofs.

While this approach has been successful for debugging solvers and providing convincing demonstrations that the fixed solvers are producing correct answers, it is important to observe that it crucially hinges on the assumption that three components are correct: (1) the low-level encoding of the problem, (2) the proof checker, and (3) the interpretation of the final output. For example, if the maximum clique solver in [GMM+20] produces a proof accepted by the VeriPB checker, then one can conclude that *if* the 0–1 ILP encoding of clique is implemented correctly, and *if* VeriPB does not contain bugs, and *if* (say) a 200-vertex graph having a maximum clique size of 13 corresponds to the optimal objective value for the low-level encoding being 187 (because it minimises the number of vertices not in the clique), then the maximum clique size is indeed 13. Such assumptions are not unreasonable—encodings have been chosen to be as simple as possible and the code can be subjected to extensive testing; the proof format is designed so that proof checking should be easy; and verifying that proof outputs correspond to solver outputs is not too cumbersome. Compared to having to trust an extremely complex solver, this is a vast improvement. However, if provably correct results are the end goal, then this still leaves much to be desired.

1.1 Our Contribution

In this work, we resolve all the concerns discussed above by presenting the first toolchain capable of end-to-end formal verification for state-of-the-art algorithms for maximum clique, subgraph isomorphism, and maximum common (connected) induced subgraph problems. Although the implementations of modern solvers for these problems are far too complicated to be formally verified by current techniques, we can still use formal verification to certify the correctness of the proof logging and proof checking process. We do so by defining a solver-friendly *augmented* VeriPB proof format; enhancing the VeriPB tool with a *proof elaborator* that can translate such augmented proofs to a more explicit *kernel* format; and designing a *formally verified proof checker* for the kernel format. This formally verified checker is also capable of providing its own formally verified encodings from graph problems to 0–1 ILPs. Finally, the output provided by the formally verified proof checker is in terms of the original problem, not the low-level encoding. This means that using the process illustrated in the right-hand part of Figure 1, if the checking process outputs (say)

```
s VERIFIED MAX CLIQUE SIZE |CLIQUE| = 13
```

then we can be absolutely sure that the maximum clique size for our graph is 13, *if* we trust the formal verification tool(s) and *if* the formal higher-order logic (HOL) specifications (as shown in Figure 2) accurately reflect what it means to be a clique. The toolchain we provide is also flexible and extensible, in that it can be readily adapted to other combinatorial problems, including problems not involving graphs.

$$
\begin{aligned}
&\text{is_clique}\ vs\ (v,e) \stackrel{\text{def}}{=} \\
&\quad vs \subseteq \{\ 0,1,\ldots,v-1\ \} \ \wedge \\
&\quad \forall x\ y.\ x \in vs \wedge y \in vs \wedge x \neq y \Rightarrow \text{is_edge}\ e\ x\ y \\
&\text{max_clique_size}\ g \stackrel{\text{def}}{=} \max_{\text{set}}\ \{\ \text{card}\ vs \mid \text{is_clique}\ vs\ g\ \}
\end{aligned}
$$

$$
\begin{aligned}
&\text{has_subgraph_iso}\ (v_p,e_p)\ (v_t,e_t) \stackrel{\text{def}}{=} \\
&\quad \exists f.\ \text{inj}\ f\ \{\ 0,1,\ldots,v_p-1\ \}\ \{\ 0,1,\ldots,v_t-1\ \} \ \wedge \\
&\qquad \forall a\ b.\ \text{is_edge}\ e_p\ a\ b \Rightarrow \text{is_edge}\ e_t\ (f\ a)\ (f\ b)
\end{aligned}
$$

Figure 2: *HOL definitions for maximum clique size of a graph with v vertices and edge set e (top), and existence of a subgraph isomorphism from a pattern graph* (v_p, e_p) *to a target graph* (v_t, e_t) *(bottom).*

1.2 Comparison to Related Work

Formally verified proof checkers have previously played an important role in SAT solving [CMS17, CHH+17, Lam20] and are vital for widespread acceptance of SAT-solver-generated mathematical proofs [HK17]. However, such proof checkers have worked only for conjunctive normal form (CNF), and only to establish that decision problems encoded in CNF are infeasible: verification that the encoding accurately reflects the problem to be solved has either been ignored or has been handled separately, e.g., [CMS19, SFL+21, CAH23]. For graph problems, previous attempts at verified proof checking have been tied to one specific problem, or even one specific algorithm, e.g., [BDM23]. In contrast, we provide formal verification for optimization problems and with much more expressive formats than CNF, and we do so in a unified way with a single pseudo-Boolean proof logging format for 0–1 linear inequalities together with a general-purpose toolchain, rather than having to design proof logging from scratch for each new combinatorial problem considered. In this way, we demonstrate that end-to-end formally verified combinatorial solving is now eminently within reach, by combining pseudo-Boolean proof logging with formally verified tools for 0–1 ILP encodings and pseudo-Boolean proof checking.

1.3 Outline of This Paper

After reviewing preliminaries in Section 2, we describe the formally verified proof checker in Section 3 and how solver proofs in a user-friendly proof format can be converted to a more restricted format accepted by this proof checker in Section 4. We report results from an experimental evaluation in Section 5. We conclude in Section 6 with a discussion of future research directions.

2 Preliminaries

Our discussion of pseudo-Boolean proof logging will be brief, since the main thrust of this work is how to formally verify proof logging rather than to design

it. See [GN21] and [BGMN23] for more on the VeriPB system and [BN21] for background on the cutting planes reasoning method used.

A *literal* ℓ over a variable x is x itself or its negation $\overline{x}$, taking values 0 (false) or 1 (true), so that $\overline{x} = 1 - x$. A *pseudo-Boolean (PB) constraint* C is a 0-1 integer linear inequality $\sum_i a_i \ell_i \geq A$, which without loss of generality we can always assume to be in *normalized form*; i.e., all literals ℓ_i are over distinct variables and the coefficients a_i and the *degree (of falsity)* A are non-negative. The *negation* $\neg C$ of C is $\sum_i a_i \overline{\ell}_i \geq \sum_i a_i - A + 1$ (saying that the sum of the coefficients of falsified literals is so large that the satisfied literals can contribute at most $A - 1$). A *pseudo-Boolean formula* is a conjunction $F = \bigwedge_j C_j$ of PB constraints.

Cutting planes [CCT87] is a method for iteratively deriving new constraints logically implied by a PB formula by taking positive linear combinations or dividing a constraint and rounding up. We say that C *unit propagates* the literal ℓ if under the current partial assignment C cannot be satisfied unless ℓ is set to true, and that C is implied by F by *reverse unit propagation (RUP)* if adding $\neg C$ to F and then unit propagating until saturation leads to contradiction in the form of a violated constraint. VeriPB allows adding constraints by RUP, which is a convenient way of avoiding having to write out explicit syntactic derivations.

In addition to deriving constraints C that are implied by F, VeriPB also has *strengthening* rules for inferring *redundant* constraints D having the property that F and $F \wedge D$ are equisatisfiable. If there is a partial mapping ω of variables to literals and/or truth values such that

$$F \cup \{\neg D\} \vdash (F \cup D)\restriction_\omega \tag{1}$$

holds, meaning that after applying ω to $F \cup \{D\}$ all of the resulting constraints can be derived by cutting planes from $F \cup \{\neg D\}$, then D can be added by *redundance-based strengthening*. There is also a similar but slightly different *dominance-based strengthening* rule. Importantly, the proof has to specify ω and also contain explicit subderivations for all *proof goals* in $(F \cup D)\restriction_\omega$ in Equation (1) unless they are obvious enough that VeriPB can automatically figure them out (e.g., by using RUP). Finally, for optimization problems there are rules to deal with objective functions and incumbent solutions, and the strengthening rules also need to be slightly adapted for this setting.

The formalization of our proof checking toolchain is carried out in the HOL4 proof assistant for classical higher-order logic [SN08]. We make particular use of the CakeML tools for production and optimization of verified CakeML source code [MO14, GMKN17] as well as for formally verified compilation [TMK+19], allowing to transfer guarantees of source-code-level correctness down to executable machine code. Where applicable, formal code snippets are pretty-printed for illustration, e.g., as shown in Figure 2. The set and first-logic notation is standard (e.g., $\Rightarrow$ denotes logical implication); other HOL notation is explained where appropriate. Formally verified results are preceded by a turnstile $\vdash$. All code is available in the supplementary material [GMM+23].

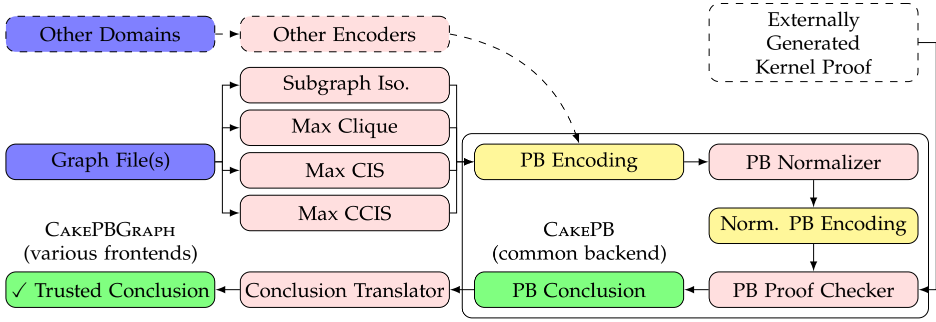


Figure 3: *Architecture of the end-to-end verified proof checkers for various graph problems.*

3 Formally Verified Graph Proof Checkers

This section details the formal verification of our pseudo-Boolean proof checker CakePB and its various graph frontends, focusing on the key architectural decisions and reusable insights behind the verification effort. An overview of the tool is shown in Figure 3. We first present the different components, and then plug them together to obtain end-to-end verified graph proof checkers.

3.1 Verified Pseudo-Boolean Proof Checking

A key design objective for CakePB is to make it a general yet effective pseudo-Boolean proof checking backend. To this end, CakePB supports a *kernel* subset of the VeriPB proof format with cutting planes, strengthening, and optimization rules as discussed in Section 2. The implementation and verification of all of this within a single proof checker backend presents several new challenges compared to prior tools for efficient verified CNF proof checking [CHH+17, Lam20, THM23]. Firstly, the pseudo-Boolean proof system features a much richer set of rules, each of which needs a formal soundness justification. Secondly, there is an intricate interplay between different proof rules, especially concerning how they preserve optimal solutions (or satisfiability for decision problems). This necessitates careful maintenance of state invariants within the proof checker implementation. And thirdly, all of the above needs to be adequately optimized for practical use, whilst being formally verified.

We use a refinement-based approach to tackle each challenge in order and at the appropriate level of abstraction.

1. The verification process starts by defining an abstract, mathematical, pseudo-Boolean semantics, with respect to which the soundness of each rule is justified. For example, we prove lemmas that justify the soundness of adding two constraints and dividing a constraint by a non-zero natural number in a cutting planes proof step:

   $\vdash$ satisfies_npbc w C_1 $\wedge$ satisfies_npbc w C_2 $\Rightarrow$
   satisfies_npbc w (add C_1 C_2)

$\vdash$ satisfies_npbc w $C \wedge k \neq 0 \Rightarrow$
    satisfies_npbc w (divide C k)

Here, satisfies_npbc w C says that the pseudo-Boolean constraint C is satisfied by the Boolean assignment w. We verify similar lemmas for all supported reasoning principles, the most involved of which is dominance-based strengthening. Specifically, this rule requires making a well-founded induction argument over an arbitrary user-specified order for Boolean assignments, for which we largely follow the proof from [BGMN23, Proposition 4].

2. Next, we implement a prototype proof checker that ensures that every application of a proof rule is valid, e.g., that divide is never applied with $k = 0$, throwing an error otherwise. The proof checker is verified to maintain key invariants on the proof state, especially the ones needed for dominance and optimization reasoning. Soundness of the checker is proved by induction over the sequence of proof steps. The main idea is illustrated by the following abridged lemma snippet.

   $\vdash$ ... $\wedge$ valid_conf *ord obj fml* $\Rightarrow$
     check_step *step ord obj fml* ... =
       Some (*ord′*,*obj′*,*fml′*, ...) $\Rightarrow$
     ... $\wedge$ valid_conf *ord′ obj′ fml′*

   Here, valid_conf *ord obj fml* says that for any satisfying assignment w to the core constraints in formula *fml*, there exists another satisfying assignment $w' \preccurlyeq w$ which satisfies all constraints in *fml*, where $\preccurlyeq$ is the order on assignments induced by *ord* and *obj*. The lemma fragment says that, whenever checking a single proof step (check_step) succeeds and returns a new proof checker state (result Some), the valid_conf invariant is maintained for the state. Other key properties verified for check_step include showing that *fml′* and *fml* are equisatisfiable by assignments that improve the best known objective value.

3. The final phase involves refining the prototype into an optimized proof checker implementation using the CakeML tools for profiling and source code verification [MO14, GMKN17]. We manually optimize several hotspots encountered in the pseudo-Boolean proofs generated in our experimental evaluation, e.g., using buffered I/O to stream large proof files, and swapping to constant-time array-based constraint lookups for cutting planes steps and hash-based proof goal coverage checks in application of the dominance-based strengthening rule.

The verified proof checker backend operates most naturally and efficiently with normalized pseudo-Boolean constraints where, in addition, variables are indexed by numbers. However, this is not the most convenient interface for frontend users. Accordingly, CakePB also includes a verified pseudo-Boolean *normalizer*. As shown in Figure 3, CakePB accepts any pseudo-Boolean formula as input (normalized or otherwise) together with an externally generated kernel proof. It produces an appropriate verified conclusion about the formula, such as satisfiability status or upper and lower bounds on the objective function, depending on the type of problem and on the claims made by the proof.

$$
\begin{aligned}
&\mathsf{is_cis}\ vs\ (v_p,e_p)\ (v_t,e_t)\ \stackrel{\text{def}}{=}\\
&\quad \exists f.\ vs \subseteq \{\ 0,1,\ldots,v_p{-}1\ \} \land \mathsf{inj}\ f\ vs\ \{\ 0,1,\ldots,v_t{-}1\ \} \land\\
&\qquad \forall a\ b.\ a \in vs \land b \in vs \Rightarrow\\
&\qquad\quad (\mathsf{is_edge}\ e_p\ a\ b \iff \mathsf{is_edge}\ e_t\ (f\ a)\ (f\ b))\\
&\mathsf{connected_subgraph}\ vs\ e\ \stackrel{\text{def}}{=}\\
&\quad \forall a\ b.\ a \in vs \land b \in vs \Rightarrow\\
&\quad\ (\lambda\, x\ y.\ y \in vs \land \mathsf{is_edge}\ e\ x\ y)^{*}\ a\ b\\
&\mathsf{is_ccis}\ vs\ (v_p,e_p)\ (v_t,e_t)\ \stackrel{\text{def}}{=}\\
&\quad \mathsf{is_cis}\ vs\ (v_p,e_p)\ (v_t,e_t) \land \mathsf{connected_subgraph}\ vs\ e_p\\
&\mathsf{max_ccis_size}\ g_p\ g_t\ \stackrel{\text{def}}{=}\\
&\quad \mathsf{max}_{\mathsf{set}}\ \{\ \mathsf{card}\ vs \mid \mathsf{is_ccis}\ vs\ g_p\ g_t\ \}
\end{aligned}
$$

$$
\begin{aligned}
&\vdash \mathsf{good_graph}\ (v_p,e_p) \land \mathsf{good_graph}\ (v_t,e_t) \land\\
&\quad \mathsf{encode}\ (v_p,e_p)\ (v_t,e_t) = \mathit{constraints} \Rightarrow\\
&\quad ((\exists vs.\ \mathsf{is_ccis}\ vs\ (v_p,e_p)\ (v_t,e_t) \land \mathsf{card}\ vs = k) \iff\\
&\qquad \exists w.\ \mathsf{satisfies}\ w\ (\mathsf{set}\ \mathit{constraints}) \land\\
&\qquad\quad \mathsf{eval_obj}\ (\mathsf{unmapped_obj}\ v_p)\ w = v_p - k)
\end{aligned}
$$

Figure 4: *HOL definition of the size of a maximum common connected induced subgraph (MCCIS) for a pattern graph g_p and a target graph g_t (top), and a correctness theorem for encoding the MCCIS problem using PB constraints (bottom).*

3.2 Verified Graph Problem Encoders

Pseudo-Boolean formulas provide a convenient format for verified frontend encoders for graph problems, which we turn to next. Graphs are represented in HOL as a pair (v,e), where v is the number of vertices corresponding to the vertex set $\{\ 0,1,\ldots,v{-}1\ \}$, and e is an edge list representation such that is_edge $e\ a\ b$ is true iff there is an edge between vertices a and b. All graphs considered here are undirected.[1] The graph encoders use a shared graph library which formalizes these basic graph notions and provides parsing functions for standard text formats such as LAD and DIMACS.

The HOL definitions of various graph problems formalized in this paper are shown in Figures 2 and 4; we use maximum common connected induced subgraph (MCCIS) as a representative example. Given a pattern graph g_p and a target graph g_t, a subset of vertices vs of g_p is a common induced subgraph (is_cis) iff there exists an injective mapping f from vs into the target graph vertices which preserves edges and non-edges. Additionally, vs is a connected subgraph of g_p iff its vertices are pairwise connected in the reflexive transitive closure (denoted *) of the induced is_edge relation. The MCCIS size is the size of the largest common connected induced subgraph between g_p and g_t (max_ccis_size).

The MCCIS pseudo-Boolean encoding from [GMM+20, Section 3.1] is implemented as a HOL function encode. The main subtlety is connected_subgraph; briefly, connectedness is encoded using additional auxiliary variables that indicate whether

[1] In practice, we apply a consistency check good_graph for undirectedness and other syntactic properties when parsing input graphs. Graphs failing the check are rejected by the encoders.

```
clique_eq_str n =def "s VERIFIED MAX CLIQUE SIZE |CLIQUE| = " ^ toString n ^ "\n"
clique_bound_str l u =def
  "s VERIFIED MAX CLIQUE SIZE BOUND " ^ toString l ^ " <= |CLIQUE| <= "
  ^ toString u ^ "\n"

1  | ⊢ cake_pb_clique_run cl fs mc ms ⇒
2  |     machine_sem mc (basis_ffi cl fs) ms ⊆
3  |       extend_with_resource_limit
4  |         { Terminate Success (cake_pb_clique_io_events cl fs) } ∧
5  |     ∃ out err. extract_fs fs (cake_pb_clique_io_events cl fs) =
6  |       Some (add_stdout (add_stderr fs err) out) ∧
7  |       (out ≠ "" ⇒
8  |         ∃ g. get_graph_dimacs fs (el 1 cl) = Some g ∧
9  |           (length cl = 2 ∧ out = concat (print_pbf (full_encode g)) ∨
10 |            length cl = 3 ∧
11 |            (out = clique_eq_str (max_clique_size g) ∨
12 |             ∃ l u. out = clique_bound_str l u ∧ (∀ vs. is_clique vs g ⇒ card vs ≤ u) ∧
13 |               ∃ vs. is_clique vs g ∧ l ≤ card vs)))
```

Figure 5: *End-to-end correctness theorem for CakePB with a maximum clique pseudo-Boolean encoder frontend.*

a walk of length n for some $n < \min(v_p, v_t)$, exists between each pair of vertices in the chosen subgraph. The correctness theorem for encode is shown in Figure 4 (bottom). It says that a CCIS of cardinality k exists iff a satisfying assignment to the encoding *constraints* exists with objective value $v_p - k$. Therefore, minimizing the objective (unmapped_obj v_p) yields the MCCIS size. Similar theorems are proved for encodings of subgraph isomorphism and maximum clique. The value of formal verification here is twofold: to gain confidence in the pen-and-paper justification of the encodings, and to ensure that the encodings are correctly implemented in code.

3.3 End-to-End Verification

Feeding the output of each frontend encoder into CakePB yields a suite of formally verified graph proof checkers, collectively called CakePBGraph. Since we are working within the CakeML ecosystem, we can further achieve *end-to-end* verification by running the CakeML compiler on CakePBGraph to transfer the source-level correctness guarantees for the CakePBGraph checkers down to the level of their respective machine code implementations.

Let us illustrate this by briefly discussing the final correctness theorem for the maximum clique proof checker as shown in Figure 5. The assumption on Line 1 is standard for all programs written in CakeML, and states that the compiled machine code is correctly loaded in memory of an x64 machine and that the appropriate command line and file system foreign function interfaces (FFIs) are available to CakeML. The first correctness guarantee on Lines 2–4 says that the code will run without crashing and will terminate safely, possibly reporting an out-of-memory resource error. The second correctness guarantee starting at Line 5–6 says there

will be (possibly empty) strings *out* and *err* printed to standard output and error, respectively. The remaining lines now claim that if standard output is non-empty, then the input file was parsed in DIMACS format to a graph g (Lines 7–8), and the output is either:

- a pretty-printed pseudo-Boolean encoding of the maximum clique problem for g (Line 9), or
- a pretty-printed conclusion string which is either:
  - a verified exact maximum clique size for g formatted using `clique_eq_str` (Line 11), or
  - verified lower and upper bounds on clique sizes in g formatted using `clique_bound_str` (Lines 12–13).

Let us clarify what needs to be trusted, or at least carefully inspected, in order to claim that the conclusions by CakePBGraph checkers are formally verified:

- The HOL definitions of the graph input parsers and of various graph problems that appear in the final correctness theorems (e.g., Figure 5). We have kept these definitions as simple as possible. Notably, the internal definitions of pseudo-Boolean semantics and cutting planes used in the proof checker are *not* part of CakePBGraph's trusted base because conversion into and out of pseudo-Boolean semantics is formally verified.
- The formal HOL model of the CakeML execution environment and its correspondence with the real system on which CakePBGraph runs. CakeML has been used in various other proof checkers, e.g., by [THM23], and its target architecture models have been validated extensively [TMK+19].
- The HOL4 theorem prover, including its logic, implementation and execution environment. The prover follows an LCF-style design [SN08] with a well-separated and trustworthy kernel responsible for checking every logical inference.

A trusted base for *binary code extraction* [KMTM18] as above is of the highest assurance standard for formally verified software—correctness is proved within a single system down to the machine code that runs. This provides a gold standard of trustworthiness for subgraph solving, in contrast to prior unverified proof checking approaches.

4 Proof Elaboration

CakePBGraph verification helps solver *users* who wish to attain a high level of trust in solver conclusions. In this section, we discuss our new *elaboration* phase, which aids solver *authors* who wish to add trustworthy proof logging and checking to their tools.

The convenience afforded by proof elaboration is illustrated in the workflow in Figure 1. First, solver authors can design their proof output with respect to

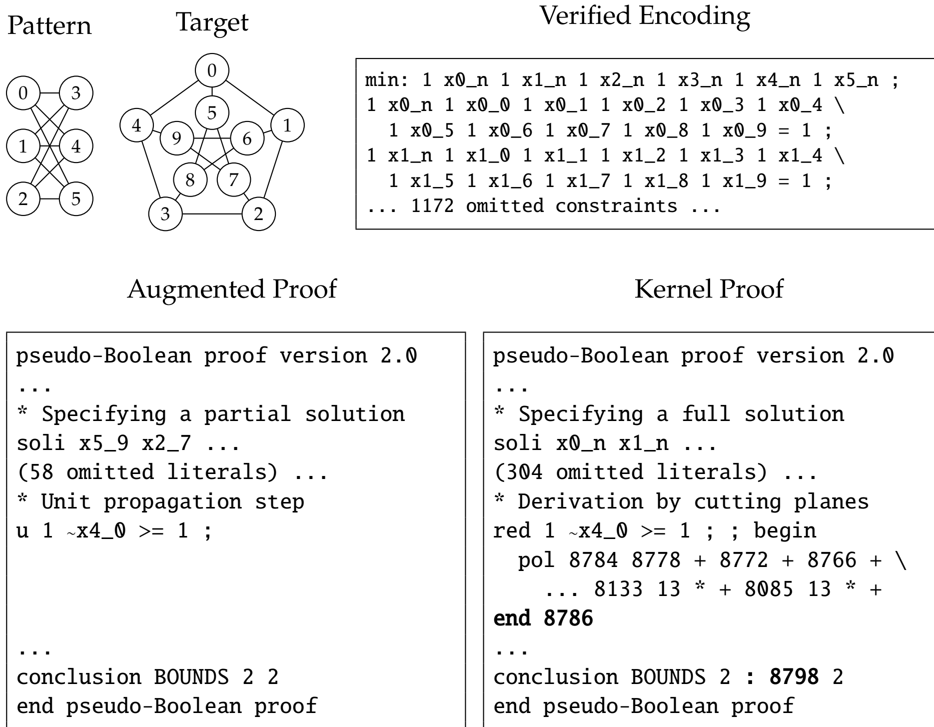


Figure 6: *(Top) MCCIS problem encoding for the pattern graph $K_{3,3}$ and the target Petersen graph. (Bottom) An augmented proof generated by a solver on the left, and a corresponding elaborated kernel proof on the right; kernel annotations in* ***bold****. When run on the kernel proof,* CakePBGraph *outputs:* `s VERIFIED MAX CCIS SIZE |CCIS| = 4`*. This corresponds to the conclusion in the proof, which claims that at least two of the six pattern vertices must be mapped to null.*

their own (untrusted) pseudo-Boolean encodings, without following the verified encodings from CakePBGraph exactly; elaboration helps to automatically line up (where possible) untrusted and verified encodings. Second, elaboration supports an *augmented* proof format with syntactic sugar that makes proof logging much easier at runtime; elaboration then fills in the necessary details to convert the proof into the kernel format understood by CakePBGraph. The VeriPB proof elaborator also performs (unverified) proof checking during the translation process, helping solver authors to detect bugs in their proof logging or solver code even before the formal verification process starts.

4.1 Lining up Encodings

Many VeriPB proof rules refer to constraints by positive integer *constraint IDs*, assigned automatically in order of appearance in the proof. It would be quite a hassle for solver authors to keep track of the exact order in which constraints in the encoding are generated by CakePBGraph. Fortunately, it is straightforward

to instead recover an ID by rederiving the constraint, which provides it with a new, known ID, before it is used. This can either be done upfront, at the start of the proof, or lazily (which avoids a potentially large overhead for instances with very short proofs). A useful fact is that the two constraints do not need to match exactly—it is sufficient that they are close enough so that VeriPB can automatically check and prove that one of them follows from the other.

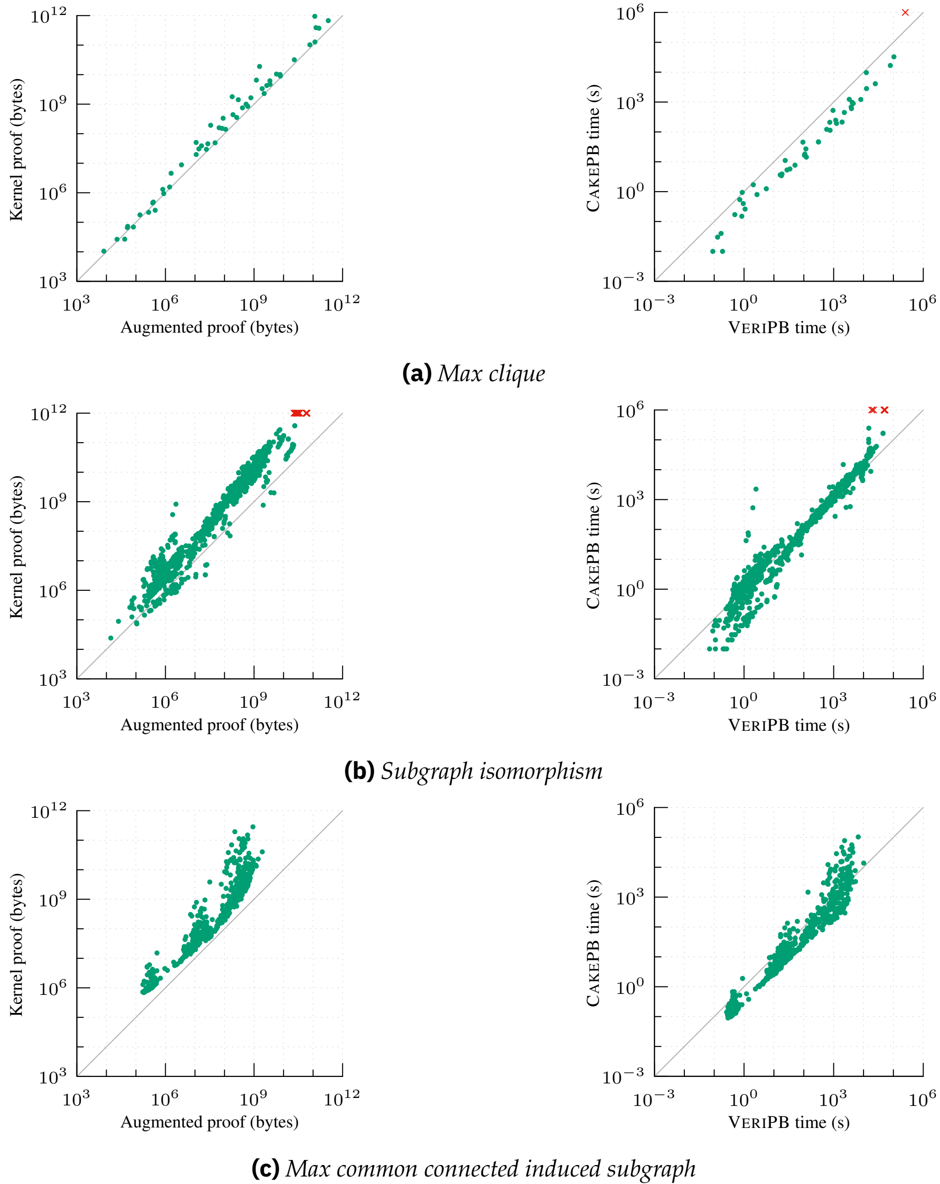
When it comes to variable names, the solver proof logging routines are required to agree exactly with the CakePBGraph encoding. This is an easier task, however, since VeriPB and CakePB both support expressive variable names. For example, for subgraph mapping problems, we use the protocol that the variable name `x1_2` means that pattern vertex 1 will be mapped to target vertex 2.

4.2 Elaborating on Syntactic Sugar

The augmented proof format contains a number of rules designed to support the ease of proof logging. Chief among these is *reverse unit propagation (RUP)*, which allows to add a constraint when the VeriPB proof checker can easily verify that it is implied by applying unit propagation. Such RUP steps occur frequently in proofs in many applications, and so have to be dealt with efficiently by the proof checker, but implementing efficient formally verified unit propagation is a challenging task even for the simpler case of CNF [FBL18]. Instead, a RUP rule application deriving C from F is converted to an explicit cutting planes proof of contradiction from $F \cup \{\neg C\}$. This is possible since unit propagation on the latter set of constraints leads to a violation (by the definition of RUP), and this in turn means that pseudo-Boolean conflict analysis can be used to derive contradiction. This algorithm is more involved than CNF-based conflict analysis as used in SAT solvers, but we employ a procedure similar to the PB conflict analysis in [EN18] for this. For optimization problems, the augmented format allows incumbent solutions to be partially specified, so long as the given assignment unit propagates to a full solution; the kernel format will always specify a full solution instead. This is illustrated in Figure 6.

Another convenient rule is *syntactic implication*, where a constraint to be derived is implied by a single (unspecified) previous constraint by simple syntactic manipulations. This condition is again easy to check, but the elaborator converts this into an explicit derivation or explicitly annotates the kernel proof with IDs. Yet another important aspect that we are ignoring here, but which is crucial for efficient proof checking, is deletion of constraints no longer needed in the proof.

Finally, applications of strengthening rules generate a separate proof goal for each constraint currently in use in the proof, which is a potentially huge overhead, but often most of these proof goals are obvious and can be skipped in the augmented format (e.g., if they can be obtained by RUP or syntactic implication). The proof elaborator fills in the necessary missing details for such proof goals.



(a) *Max clique*

(b) *Subgraph isomorphism*

(c) *Max common connected induced subgraph*

Figure 7: *Experiments using the Glasgow Subgraph Solver on (a) max clique, (b) subgraph isomorphism, and (c) max common connected induced subgraph problem instances. In the left column, comparisons of kernel and augmented proof sizes; in the right column, time comparisons for verified and unverified checking of kernel and augmented proofs, respectively. Crosses indicate failures due to space or memory limits.*

5 Experiments

To validate our approach, we performed experiments on a cluster of machines with dual AMD EPYC 7643 processors, 2TBytes RAM, and a RAID array of solid state drives, running Ubuntu 22.04. We ran up to 40 jobs in parallel, and limited each individual process to 64GBytes RAM. Note that performance of the verification process is strongly affected by I/O and memory cache speeds, and so we do not expect running time measurements to be highly reproducible, but they should still be indicative of the feasibility of the approach and the slowdowns that one might encounter. We used the Glasgow Subgraph Solver [MPT20] as the proof-producing solver for all experiments, and made small modifications so that it would lazily recover constraint IDs as required. The results are plotted on an instance by instance basis in Figure 7 and explained below.

For maximum clique, we took the 54 instances from the Second DIMACS Implementation Challenge [JT96] that [GMM+20] were able to check. We managed to produce proofs for and formally verify 50 of these instances; for the 4 instances that we could not verify, 3 were due to VeriPB taking over one week to check the proof files, and the final one to the 64GByte memory limit for the verified checker. Over the successfully checked instances, translating augmented proofs to kernel proofs took, on average, 18% longer than simply verifying the proofs, and produced proof files that were on average 2.26 times as large. However, verified checking of these kernel proofs was consistently faster than checking the original augmented proofs using VeriPB: the average running time was 3.8 times lower.

For subgraph isomorphism, we used the same subset of 1,226 small-to-medium-sized instances from the benchmark set in [KMS16] as was studied by [GMN20]. We were able to verify 417 satisfiable and 784 unsatisfiable instances; 13 instances failed due to memory limits on the verified checker, and 12 instances when the converted kernel proofs exceeded 500GBytes in size. Performance-wise, running VeriPB and asking it to output a kernel proof was on average 27% slower than verification alone. Producing the verified encoding was never a significant cost in the process. Verifying kernel proofs was on average 2.4 times slower than verifying the original, augmented proofs; the former were on average 10.5 times larger than the latter.

For maximum common connected induced subgraph, we used a database of randomly generated instances [CFV07, DFSV03], and ran the solver in clique reformulation mode. We were able to verify all 690 instances involving up to 20 vertices in each graph. Elaborating the proofs took on average 43% longer than verifying them using VeriPB, and the proofs were on average 14.7 times larger. However, verifying the kernel proofs using CakePB took on average only 9% longer than using VeriPB for the original, augmented proofs.

Across each problem family, producing formally verified encodings was always extremely cheap, and asking VeriPB to produce an elaborated kernel proof was never substantially more expensive than simply checking the augmented proof. This is to be expected: VeriPB already has to produce nearly all of the information needed for proof elaboration to check a proof anyway. Checking elaborated proofs was sometimes a little faster than checking the original, augmented proof, and

sometimes a little slower, and we were able to formally check almost every proof that was amenable to unverified checking.

6 Conclusion

In this paper, we present the first efficient toolchain for formal end-to-end verification of state-of-the-art subgraph solving. Our design is easily adaptable, which opens up the possibility of bringing formal verification to other combinatorial problem domains where problem instances can be suitably represented using the expressivity of 0–1 integer linear programs. In fact, our formally verified CakePB proof checker equipped with a CNF frontend has also been used for SAT solving in the SAT Competition 2023 [BMM+23], supporting, also for the first time, efficient verified proof logging and checking for the full range of advanced techniques used in modern SAT solvers such as cardinality reasoning, Gaussian elimination, and symmetry breaking. A future challenge of particular interest would be to provide a formally verified setting for the proof logging techniques for constraint programming developed in a sequence of papers by [EGMN20, GMN22] and [MM23]. It would also be valuable to expand the reach of pseudo-Boolean proof logging to problems like (projected) model enumeration problems, which were dealt with in a somewhat ad-hoc fashion by [GMM+20].

To further improve performance, it would be highly desirable to enhance the VeriPB elaborator with *proof trimming* to be able to remove unnecessary proof steps before handing the kernel proof to CakePB. Currently, our system verifies all of the steps carried out by the solver to reach its conclusion. This is useful for detecting solver bugs, but for storing and distributing proofs a trimmed proof would suffice and could be much faster to verify. Another significant source of performance gains could come from switching from a text proof format to a binary format: although this would lose some human-readability, our experiments suggest that text parsing often forms a substantial portion of the elaboration and checking times.

Acknowledgements

Stephan Gocht and Jakob Nordström were supported by the Swedish Research Council grant 2016-00782, and Jakob Nordström also received funding from the Independent Research Fund Denmark grant 9040-00389B. Ciaran McCreesh was supported by a Royal Academy of Engineering research fellowship, and by the Engineering and Physical Sciences Research Council [grant number EP/X030032/1]. Magnus Myreen was supported by Swedish Research Council grant 2021-05165. Andy Oertel was supported by the Wallenberg AI, Autonomous Systems and Software Program (WASP) funded by the Knut and Alice Wallenberg Foundation. Yong Kiam Tan was supported by A*STAR, Singapore. Part of this work was carried out while taking part in the Dagstuhl workshops 22411 "Theory and Practice of SAT and Combinatorial Solving" and 23261 "SAT Encodings and Beyond", as well as in the extended reunion of the program "Satisfiability: Theory, Practice, and

Beyond" in the spring of 2023 at the Simons Institute for the Theory of Computing at UC Berkeley.

References

[AGJ+18] Özgür Akgün, Ian P. Gent, Christopher Jefferson, Ian Miguel, and Peter Nightingale. Metamorphic testing of constraint solvers. In *Proceedings of the 24th International Conference on Principles and Practice of Constraint Programming (CP '18)*, volume 11008 of *Lecture Notes in Computer Science*, pages 727–736. Springer, August 2018.

[BDM23] Milan Bankovic, Ivan Drecun, and Filip Maric. A proof system for graph (non)-isomorphism verification. *Logical Methods in Computer Science*, 19(1), February 2023.

[BGMN23] Bart Bogaerts, Stephan Gocht, Ciaran McCreesh, and Jakob Nordström. Certified dominance and symmetry breaking for combinatorial optimisation. *Journal of Artificial Intelligence Research*, 77:1539–1589, August 2023. Preliminary version in *AAAI '22*.

[BHvMW21] Armin Biere, Marijn J. H. Heule, Hans van Maaren, and Toby Walsh, editors. *Handbook of Satisfiability*, volume 336 of *Frontiers in Artificial Intelligence and Applications*. IOS Press, 2nd edition, February 2021.

[BMM+23] Bart Bogaerts, Ciaran McCreesh, Magnus O. Myreen, Jakob Nordström, Andy Oertel, and Yong Kiam Tan. Documentation of VeriPB and CakePB for the SAT competition 2023. Available at `https://satcompetition.github.io/2023/checkers.html`, March 2023.

[BMN22] Bart Bogaerts, Ciaran McCreesh, and Jakob Nordström. Solving with provably correct results: Beyond satisfiability, and towards constraint programming. Tutorial at the *28th International Conference on Principles and Practice of Constraint Programming*. Slides available at `http://www.jakobnordstrom.se/presentations/`, August 2022.

[BN21] Samuel R. Buss and Jakob Nordström. Proof complexity and SAT solving. In Biere et al. [BHvMW21], chapter 7, pages 233–350.

[BR07] Robert Bixby and Edward Rothberg. Progress in computational mixed integer programming—A look back from the other side of the tipping point. *Annals of Operations Research*, 149(1):37–41, February 2007.

[CAH23] Cayden R. Codel, Jeremy Avigad, and Marijn J. H. Heule. Verified encodings for SAT solvers. In *Proceedings of the 23rd Conference on Formal Methods in Computer-Aided Design (FMCAD '23)*, pages 141–151, October 2023.

[CCT87] William Cook, Collette Rene Coullard, and György Turán. On the complexity of cutting-plane proofs. *Discrete Applied Mathematics*, 18(1):25–38, November 1987.

[CFV07] Donatello Conte, Pasquale Foggia, and Mario Vento. Challenging complexity of maximum common subgraph detection algorithms: A performance analysis of three algorithms on a wide database of graphs. *Journal of Graph Algorithms and Applications*, 11(1):99–143, January 2007.

[CHH+17] Luís Cruz-Filipe, Marijn J. H. Heule, Warren A. Hunt Jr., Matt Kaufmann, and Peter Schneider-Kamp. Efficient certified RAT verification. In *Proceedings of the 26th International Conference on Automated Deduction (CADE-26)*, volume 10395 of *Lecture Notes in Computer Science*, pages 220–236. Springer, August 2017.

[CKSW13] William Cook, Thorsten Koch, Daniel E. Steffy, and Kati Wolter. A hybrid branch-and-bound approach for exact rational mixed-integer programming. *Mathematical Programming Computation*, 5(3):305–344, September 2013.

[CMS17] Luís Cruz-Filipe, João P. Marques-Silva, and Peter Schneider-Kamp. Efficient certified resolution proof checking. In *Proceedings of the 23rd International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS '17)*, volume 10205 of *Lecture Notes in Computer Science*, pages 118–135. Springer, April 2017.

[CMS19] Luís Cruz-Filipe, João P. Marques-Silva, and Peter Schneider-Kamp. Formally verifying the solution to the Boolean Pythagorean triples problem. *Journal of Automated Reasoning*, 63(3):695–722, October 2019.

[DFSV03] Massimo De Santo, Pasquale Foggia, Carlo Sansone, and Mario Vento. A large database of graphs and its use for benchmarking graph isomorphism algorithms. *Pattern Recognition Letters*, 24(8):1067–1079, May 2003.

[EGMN20] Jan Elffers, Stephan Gocht, Ciaran McCreesh, and Jakob Nordström. Justifying all differences using pseudo-Boolean reasoning. In *Proceedings of the 34th AAAI Conference on Artificial Intelligence (AAAI '20)*, pages 1486–1494, February 2020.

[EN18] Jan Elffers and Jakob Nordström. Divide and conquer: Towards faster pseudo-Boolean solving. In *Proceedings of the 27th International Joint Conference on Artificial Intelligence (IJCAI '18)*, pages 1291–1299, July 2018.

[FBL18] Mathias Fleury, Jasmin Christian Blanchette, and Peter Lammich. A verified SAT solver with watched literals using imperative HOL. In *Proceedings of the 7th ACM SIGPLAN International Conference on Certified Programs and Proofs (CPP '18)*, pages 158—171, January 2018.

[GMKN17] Armaël Guéneau, Magnus O. Myreen, Ramana Kumar, and Michael Norrish. Verified characteristic formulae for CakeML. In *Proceedings of the 26th European Symposium on Programming (ESOP '17)*, volume 10201 of *Lecture Notes in Computer Science*, pages 584–610. Springer, April 2017.

[GMM+20] Stephan Gocht, Ross McBride, Ciaran McCreesh, Jakob Nordström, Patrick Prosser, and James Trimble. Certifying solvers for clique and maximum common (connected) subgraph problems. In *Proceedings of the 26th International Conference on Principles and Practice of Constraint Programming (CP '20)*, volume 12333 of *Lecture Notes in Computer Science*, pages 338–357. Springer, September 2020.

[GMM+23] Stephan Gocht, Ciaran McCreesh, Magnus O. Myreen, Jakob Nordström, Andy Oertel, and Yong Kiam Tan. End-to-end verification for subgraph solving: Supplementary material. `https://doi.org/10.5281/zenodo.10369401`, December 2023.

[GMN20] Stephan Gocht, Ciaran McCreesh, and Jakob Nordström. Subgraph isomorphism meets cutting planes: Solving with certified solutions. In *Proceedings of the 29th International Joint Conference on Artificial Intelligence (IJCAI '20)*, pages 1134–1140, July 2020.

[GMN22] Stephan Gocht, Ciaran McCreesh, and Jakob Nordström. An auditable constraint programming solver. In *Proceedings of the 28th International Conference on Principles and Practice of Constraint Programming (CP '22)*, volume 235 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 25:1–25:18, August 2022.

[GN21] Stephan Gocht and Jakob Nordström. Certifying parity reasoning efficiently using pseudo-Boolean proofs. In *Proceedings of the 35th AAAI Conference on Artificial Intelligence (AAAI '21)*, pages 3768–3777, February 2021.

[GSD19] Xavier Gillard, Pierre Schaus, and Yves Deville. SolverCheck: Declarative testing of constraints. In *Proceedings of the 25th International Conference on Principles and Practice of Constraint Programming (CP '19)*, volume 11802 of *Lecture Notes in Computer Science*, pages 565–582. Springer, October 2019.

[GSVW14] Maria Garcia de la Banda, Peter J. Stuckey, Pascal Van Hentenryck, and Mark Wallace. The future of optimization technology. *Constraints*, 19(2):126–138, April 2014.

[HHW13a] Marijn J. H. Heule, Warren A. Hunt Jr., and Nathan Wetzler. Trimming while checking clausal proofs. In *Proceedings of the 13th International Conference on Formal Methods in Computer-Aided Design (FMCAD '13)*, pages 181–188, October 2013.

[HHW13b] Marijn J. H. Heule, Warren A. Hunt Jr., and Nathan Wetzler. Verifying refutations with extended resolution. In *Proceedings of the 24th International Conference on Automated Deduction (CADE-24)*, volume 7898 of *Lecture Notes in Computer Science*, pages 345–359. Springer, June 2013.

[HK17] Marijn J. H. Heule and Oliver Kullmann. The science of brute force. *Communications of the ACM*, 60(8):70–79, August 2017.

[JT96] David S. Johnson and Michael A. Trick. Introduction to the second DIMACS challenge: Cliques, coloring, and satisfiability. In *Cliques, Coloring and Satisfiability: Second DIMACS Implementation Challenge*, volume 26 of *DIMACS Series in Discrete Mathematics and Theoretical Computer Science*, pages 1–10. American Mathematical Society, 1996.

[KMS16] Lars Kotthoff, Ciaran McCreesh, and Christine Solnon. Portfolios of subgraph isomorphism algorithms. In *10th International Conference on Learning and Intelligent Optimization (LION '16), Selected Revised Papers*, volume 10079 of *Lecture Notes in Computer Science*, pages 107–122. Springer, May-June 2016.

[KMTM18] Ramana Kumar, Eric Mullen, Zachary Tatlock, and Magnus O. Myreen. Software verification with ITPs should use binary code extraction to reduce the TCB. In *Proceedings of the 9th International Conference on Interactive Theorem Proving (ITP '18)*, volume 10895 of *Lecture Notes in Computer Science*, pages 362–369. Springer, July 2018.

[Lam20] Peter Lammich. Efficient verified (UN)SAT certificate checking. *Journal of Automated Reasoning*, 64(3):513–532, March 2020. Extended version of paper in *CADE* 2017.

[MM23] Matthew McIlree and Ciaran McCreesh. Proof logging for smart extensional constraints. In *Proceedings of the 29th International Conference on Principles and Practice of Constraint Programming (CP '23)*, volume 280 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 26:1–26:17, August 2023.

[MMNS11] Ross M. McConnell, Kurt Mehlhorn, Stefan Näher, and Pascal Schweitzer. Certifying algorithms. *Computer Science Review*, 5(2):119–161, May 2011.

[MO14] Magnus O. Myreen and Scott Owens. Proof-producing translation of higher-order logic into pure and stateful ML. *Journal of Functional Programming*, 24(2–3):284–315, January 2014.

[MPT20] Ciaran McCreesh, Patrick Prosser, and James Trimble. The Glasgow subgraph solver: Using constraint programming to tackle hard subgraph isomorphism problem variants. In *Proceedings of the 13th International Conference on Graph Transformation (ICGT '20)*, volume 12150 of *Lecture Notes in Computer Science*, pages 316–324. Springer, June 2020.

[SFL+21] Xiaomu Shi, Yu-Fu Fu, Jiaxiang Liu, Ming-Hsien Tsai, Bow-Yaw Wang, and Bo-Yin Yang. CoqQFBV: A scalable certified SMT quantifier-free bit-vector solver. In *Proceedings of the 33rd International Conference on Computer Aided Verification (CAV '21)*, volume 12760 of *Lecture Notes in Computer Science*, pages 149–171. Springer, July 2021.

[SN08] Konrad Slind and Michael Norrish. A brief overview of HOL4. In *Proceedings of the 21st International Conference on Theorem Proving in Higher Order Logics (TPHOLs '08)*, volume 5170 of *Lecture Notes in Computer Science*, pages 28–32. Springer, August 2008.

[THM23] Yong Kiam Tan, Marijn J. H. Heule, and Magnus O. Myreen. Verified propagation redundancy and compositional UNSAT checking in CakeML. *International Journal on Software Tools for Technology Transfer*, 25:167–184, February 2023. Preliminary version in *TACAS '21*.

[TMK+19] Yong Kiam Tan, Magnus O. Myreen, Ramana Kumar, Anthony C. J. Fox, Scott Owens, and Michael Norrish. The verified CakeML compiler backend. *Journal of Functional Programming*, 29:e2:1–e2:57, February 2019.

[WHH14] Nathan Wetzler, Marijn J. H. Heule, and Warren A. Hunt Jr. DRAT-trim: Efficient checking and trimming using expressive clausal proofs. In *Proceedings of the 17th International Conference on Theory and Applications of Satisfiability Testing (SAT '14)*, volume 8561 of *Lecture Notes in Computer Science*, pages 422–429. Springer, July 2014.

Certified MaxSAT Preprocessing

Abstract

Building on the progress in Boolean satisfiability (SAT) solving over the last decades, maximum satisfiability (MaxSAT) has become a viable approach for solving NP-hard optimization problems. However, ensuring correctness of MaxSAT solvers has remained a considerable concern. For SAT, this is largely a solved problem thanks to the use of proof logging, meaning that solvers emit machine-verifiable proofs to certify correctness. However, for MaxSAT, proof logging solvers have started being developed only very recently. Moreover, these nascent efforts have only targeted the core solving process, ignoring the preprocessing phase where input problem instances can be substantially reformulated before being passed on to the solver proper.

In this work, we demonstrate how pseudo-Boolean proof logging can be used to certify the correctness of a wide range of modern MaxSAT preprocessing techniques. By combining and extending the VeriPB and CakePB tools, we provide formally verified end-to-end proof checking that the input and preprocessed output MaxSAT problem instances have the same optimal value. An extensive evaluation on applied MaxSAT benchmarks shows that our approach is feasible in practice.

1 Introduction

The development of Boolean satisfiability (SAT) solvers is arguably one of the true success stories of modern computer science—today, SAT solvers are routinely used as core engines in many types of complex automated reasoning systems. One example of this is SAT-based optimization, usually referred to as *maximum satisfiability (MaxSAT) solving.* The improved performance of SAT solvers, coupled with increasingly sophisticated techniques for using SAT solver calls to reason

Hannes Ihalainen, Andy Oertel, Yong Kiam Tan, Jeremias Berg, Matti Järvisalo, Magnus O. Myreen, and Jakob Nordström. "Certified MaxSAT Preprocessing". In *Proceedings of the 12th International Joint Conference on Automated Reasoning (IJCAR '24)*, volume 14739 of *Lecture Notes in Computer Science*, pages 396–418. Springer, July 2024.

about optimization problems, have made MaxSAT solvers a powerful tool for tackling real-world NP-hard optimization problems [BHvMW21].

However, Modern MaxSAT solvers are quite intricate pieces of software, and it has been shown repeatedly in the MaxSAT evaluations [Maxb] that even the best solvers sometimes report incorrect results. This was previously a serious issue also for SAT solvers (see, e.g., [BLB10]), but the SAT community has essentially eliminated this problem by requiring that solvers should be *certifying* [ABM+11, MMNS11], i.e., not only report whether a given formula is satisfiable or unsatisfiable but also produce a machine-verifiable proof that this conclusion is correct. Many different SAT proof formats such as RUP [GN03], TraceCheck [Bie06], GRIT [CMS17], and LRAT [CHH+17] have been proposed, with DRAT [HHW13a, HHW13b, WHH14] established as the de facto standard; for the last ten years, proof logging has been compulsory in the (main track of the) SAT competitions [SAT]. It is all the more striking, then, that until recently no similar developments have been observed in MaxSAT solving.

1.1 Previous Work

A first natural question to ask—since MaxSAT solvers are based on repeated calls to SAT solvers—is why we cannot simply use SAT proof logging also for MaxSAT. The problem is that DRAT can only reason about clauses, whereas MaxSAT solvers argue about costs of solutions and values of objective functions. Translating such claims to clausal form would require an external tool to certify correctness of the translation. Also, such clausal translations incur a significant overhead and do not seem well-adapted for, e.g., counting arguments in MaxSAT.

While there have been several attempts to design proof systems specifically for MaxSAT solving [BLM07, FMSV20, IBJ22, LNOR11, MIB+19, MM11, PCH20, PCH21, PCH22], none of these have come close to providing a general proof logging solution, because they apply only for very specific algorithm implementations and/or fail to capture the full range of techniques used. Recent papers have instead proposed using pseudo-Boolean proof logging with VeriPB [BGMN23, GN21] to certify correctness of so-called solution-improving solvers [VDB22] and core-guided solvers [BBN+23]. Although these works demonstrate, for the first time, practical proof logging for modern MaxSAT solving, the methods developed thus far only apply to the core solving process. This ignores the preprocessing phase, where the input formula can undergo major reformulation. State-of-the-art solvers sometimes use stand-alone preprocessor tools, or sometimes integrate preprocessing-style reasoning more tightly within the MaxSAT solver engine, to speed up the search for optimal solutions. Some of these preprocessing techniques are lifted from SAT to MaxSAT, but there are also native MaxSAT preprocessing methods that lack analogies in SAT solving.

1.2 Our Contribution

In this paper, we show, for the first time, how to use pseudo-Boolean proof logging with VeriPB to produce proofs of correctness for a wide range of preprocessing techniques used in modern MaxSAT solvers. VeriPB proof logging

has previously been successfully used not only for core MaxSAT search as discussed above, but also for advanced SAT solving techniques (including symmetry breaking) [BGMN23, GMNO22, GN21], subgraph solving [GMM+20, GMM+24, GMN20], constraint programming [EGMN20, GMN22, MM23, MMN24], and 0–1 ILP presolving [HOGN24], and we add MaxSAT preprocessing to this list.

In order to do so, we extend the VeriPB proof format to include an *output section* where a reformulated output can be presented, and where the pseudo-Boolean proof establishes that this output formula and the input formula are *equioptimal*, i.e., have optimal solutions of the same value. We also enhance CakePB [BMM+23, GMM+24]—a verified proof checker for pseudo-Boolean proofs—to handle proofs of reformulation. In this way, we obtain an end-to-end formally verified toolchain for certified preprocessing of MaxSAT instances.

It is worth noting that although preprocessing is also a critical component in SAT solving, we are not aware of any tool for certifying reformulations even for the restricted case of decision problems, i.e., showing that formulas are *equisatisfiable*—the DRAT format and tools support proofs that satisfiability of an input CNF formula F implies satisfiability of an output CNF formula G but not the converse direction (except in the special case where F is a subset of G). To the best of our knowledge, our work presents the first practical tool for proving (two-way) equisatisfiability or equioptimality of reformulated problems.

We have performed computational experiments running a MaxSAT preprocessor with proof logging and proof checking on benchmarks from the MaxSAT evaluations [Maxb]. Although there is certainly room for improvements in performance, these experiments provide empirical evidence for the feasibility of certified preprocessing for real-world MaxSAT benchmarks.

1.3 Organization of This Paper

After reviewing preliminaries in Section 2, we explain our pseudo-Boolean proof logging for MaxSAT preprocessing in Section 3, and Section 4 discusses verified proof checking. We present results from a computational evaluation in Section 5, after which we conclude with a summary and outlook for future work in Section 6.

2 Preliminaries

We write ℓ to denote a literal, i.e., a $\{0,1\}$-valued Boolean variable x or its negation $\overline{x} = 1 - x$. A *clause* $C = \ell_1 \vee \ldots \vee \ell_k$ is a disjunction of literals, where a *unit clause* consists of only one literal. A formula in *conjunctive normal form (CNF)* $F = C_1 \wedge \ldots \wedge C_m$ is a conjunction of clauses, where we think of clauses and formulas as sets so that there are no repetitions and order is irrelevant.

A *pseudo-Boolean (PB) constraint* is a 0–1 linear inequality $\sum_j a_j \ell_j \geq b$, where, when convenient, we can assume all literals ℓ_j to refer to distinct variables and all integers a_j and b to be positive (so-called *normalized form*). A *pseudo-Boolean formula* is a conjunction of such constraints. We identify the clause $C = \ell_1 \vee \cdots \vee \ell_k$ with the pseudo-Boolean constraint $\mathrm{PB}(C) = \ell_1 + \cdots + \ell_k \geq 1$, so a CNF formula F is just a special type of PB formula $\mathrm{PB}(F) = \{\mathrm{PB}(C) \mid C \in F\}$.

A *(partial) assignment* ρ mapping variables to $\{0, 1\}$, is extended to literals by respecting the meaning of negation, satisfies a PB constraint $\sum_j a_j \ell_j \geq b$ if $\sum_{\ell_j : \rho(\ell_j)=1} a_j \geq b$ (assuming normalized form). A PB formula is satisfied by ρ if all constraints in it are. We also refer to total satisfying assignments ρ as *solutions*. In a *pseudo-Boolean optimization (PBO)* problem we ask for a solution minimizing a given *objective* function $O = \sum_j c_j \ell_j + W$, where c_j and W are integers and W represents a trivial lower bound on the minimum cost.

2.1 Pseudo-Boolean Proof Logging Using Cutting Planes

The pseudo-Boolean proof logging in VeriPB is based on the *cutting planes* proof system [CCT87] with extensions as discussed briefly next. We refer the reader to [BN21] for and in-depth discussion of cutting planes and to [BGMN23, Goc22, HOGN24, Ver] for more detailed information about the VeriPB proof system and format.

A pseudo-Boolean proof maintains two sets of *core constraints* $\mathcal{C}$ and *derived constraints* $\mathcal{D}$ under which the objective O should be minimized. At the start of the proof, $\mathcal{C}$ is initialized to the constraints in the input formula F. Any constraints derived by the rules described below are placed in $\mathcal{D}$, from where they can later be moved to $\mathcal{C}$ (but not vice versa). The proof system semantics preserves the invariant that the optimal value of any solution to $\mathcal{C}$ and to the original input problem F is the same. New constraints can be derived from $\mathcal{C} \cup \mathcal{D}$ by performing *addition* of two constraints or *multiplication* of a constraint by a positive integer, and *literal axioms* $\ell \geq 0$ can be used at any time. Additionally, we can apply *division* to $\sum_j a_j \ell_j \geq b$ by a positive integer d followed by rounding up to obtain $\sum_j \lceil a_j/d \rceil \ell_j \geq \lceil b/d \rceil$, and *saturation* to yield $\sum_j \min\{a_j, b\} \cdot \ell_j \geq b$ (where we again assume normalized form).

The negation of a constraint $C = \sum_j a_j \ell_j \geq b$ is $\neg C = \sum_j a_j \ell_j \leq b - 1$. For a (partial) assignment ρ we write $C\restriction_\rho$ for the *restricted constraint* obtained by replacing literals in C assigned by ρ with their values and simplifying. We say that C *unit propagates* ℓ *under* ρ if $C\restriction_\rho$ cannot be satisfied unless ℓ is assigned to 1. If repeated unit propagation on all constraints in $\mathcal{C} \cup \mathcal{D} \cup \{\neg C\}$, starting with the empty assignment $\rho = \emptyset$, leads to contradiction in the form of an unsatisfiable constraint, we say that C follows by *reverse unit propagation (RUP)* from $\mathcal{C} \cup \mathcal{D}$. Such (efficiently verifiable) RUP steps are allowed in VeriPB proofs as a convenient way to avoid writing out an explicit cutting planes derivation. We use the same notation $C\restriction_\omega$ to denote the result of applying to C a *(partial) substitution* ω, which can map variables not only to $\{0, 1\}$ but also to literals, and extend this notation to sets of constraints by taking unions.

In addition to the above rules, which derive semantically implied constraints, there is a *redundance-based strengthening rule*, or just *redundance rule* for short, that can derive non-implied constraints C as long as they do not change the feasibility or optimal value. This can be guaranteed by exhibiting a *witness substitution* ω such that for any total assignment α satisfying $\mathcal{C} \cup \mathcal{D}$ but violating C, the composition $\alpha \circ \omega$ is another total assignment that satisfies $\mathcal{C} \cup \mathcal{D} \cup \{C\}$ and yields an objective value that is at least as good. Formally, C can be derived from $\mathcal{C} \cup \mathcal{D}$ by exhibiting

ω and subproofs for

$$\mathcal{C} \cup \mathcal{D} \cup \{\neg C\} \vdash (\mathcal{C} \cup \mathcal{D} \cup \{C\})\restriction_\omega \cup \{O \geq O\restriction_\omega\}\,, \tag{1}$$

using the previously discussed rules (where the notation $\mathcal{C}_1 \vdash \mathcal{C}_2$ means that the constraints $\mathcal{C}_2$ can be derived from the constraints $\mathcal{C}_1$).

During preprocessing, constraints in the input formula are often deleted or replaced by other constraints, in which case the proof should establish that these deletions maintain equioptimality. Removing constraints from the derived set $\mathcal{D}$ is unproblematic, but unrestricted deletion from the core set $\mathcal{C}$ can clearly introduce spurious better solutions. Therefore, removing C from $\mathcal{C}$ can only be done by the *checked deletion rule*, which requires a proof that the redundance rule can be used to rederive C from $\mathcal{C} \setminus \{C\}$ (see [BGMN23] for a more detailed explanation).

Finally, it turns out to be useful to allow replacing O by a new objective O' using an *objective function update rule*, as long as this does not change the optimal value of the problem. Formally, updating the objective from O to O' requires derivations of the two constraints $O \geq O'$ and $O' \geq O$ from the core set $\mathcal{C}$, which shows that any satisfying solution to $\mathcal{C}$ has the same value for both objectives. More details on this rule can be found in [HOGN24].

2.2 Maximum Satisfiability

A WCNF instance of (weighted partial) maximum satisfiability $\mathcal{F}^W = (F_H, F_S)$ is a conjunction of two CNF formulas F_H and F_S with *hard* and *soft* clauses, respectively, where soft clauses $C \in F_S$ have positive weights w^C. A solution ρ to $\mathcal{F}^W$ must satisfy F_H and has value $\textsc{cost}(F_S, \rho)$ equal to the sum of weights of all soft clauses not satisfied by ρ. The optimum $\textsc{opt}(\mathcal{F}^W)$ of $\mathcal{F}^W$ is the minimum of $\textsc{cost}(F_S, \rho)$ over all solutions ρ, or ∞ if no solution exists.

State-of-the-art MaxSAT preprocessors such as MaxPre [IBJ22, KBSJ17] take a slightly different *objective-centric* view [BJ19] of MaxSAT instances $\mathcal{F} = (F, O)$ as consisting of a CNF formula F and an objective function $O = \sum_j c_j \ell_j + W$ to be minimized under assignments ρ satisfying F. A WCNF MaxSAT instance $\mathcal{F}^W = (F_H, F_S)$ is converted into objective-centric form $\textsc{ObjMaxSAT}(\mathcal{F}^W) = (F, O)$ by letting the formula $F = F_H \cup \{C \vee b_C \mid C \in F_S, |C| > 1\}$ of $\textsc{ObjMaxSAT}(\mathcal{F}^W)$ consist of the hard clauses of $\mathcal{F}^W$ and the non-unit soft clauses in F_S, each extended with a fresh variable b_C that does not appear in any other clause. The objective $O = \sum_{(\bar{\ell}) \in F_S} w^{(\bar{\ell})} \ell + \sum w^C b_C$ contains literals ℓ for all unit soft clauses $\bar{\ell}$ in F_S as well as literals for all new variables b_C, with coefficients equal to the weights of the corresponding soft clauses. In other words, each unit soft clause $\bar{\ell} \in F_S$ of weight w is transformed into the term $w \cdot \ell$ in the objective function O, and each non-unit soft clause C is transformed into the hard clause $C \vee b_C$ paired with the unit soft clause $(\bar{b}_C)$ with same weight as C. The following observation summarizes the properties of $\textsc{ObjMaxSAT}(\mathcal{F}^W)$ that are central to our work.

Observation 1. *For any solution ρ to a WCNF MaxSAT instance $\mathcal{F}^W$ there exists a solution ρ' to $(F, O) = \textsc{ObjMaxSAT}(\mathcal{F}^W)$ with $O(\rho') = \textsc{cost}(\mathcal{F}^W, \rho)$. Conversely, if ρ' is a solution to $\textsc{ObjMaxSAT}(\mathcal{F}^W)$, then there exists a solution ρ of $\mathcal{F}^W$ for which $\textsc{cost}(\mathcal{F}^W, \rho) \leq O(\rho')$.*

For the second part of the observation, the reason $O(\rho')$ is only an upper bound on $\textsc{cost}(\mathcal{F}^W, \rho)$ is that the encoding forces b_C to be true whenever C is not satisfied by an assignment but not vice versa.

An objective-centric MaxSAT instance (F, O), in turn, clearly has the same optimum as the pseudo-Boolean optimization problem of minimizing O subject to $\mathrm{PB}(F)$. For the end-to-end formal verification, the fact that this coincides with $\textsc{opt}(\mathcal{F}^W)$ needs to be formalized into theorems as shown in Figure 4.

3 Proof Logging for MaxSAT Preprocessing

We now discuss how pseudo-Boolean proof logging can be used to reason about correctness of MaxSAT preprocessing steps. Our approach maintains the invariant that the current working instance in the preprocessor is synchronized with the PB constraints in the core set $\mathcal{C}$ as described in Section 2.2. At the end of each preprocessing step (i.e., application of a preprocessing technique) the set of derived constraints $\mathcal{D}$ is empty. All constraints derived in the proof as described in this section are moved to the core set, and constraints are always removed by checked deletion from the core set. Full technical details are in Appendix A.

3.1 Overview

All our preprocessing steps maintain *equioptimality*, which means that if preprocessing of the WCNF MaxSAT instance $\mathcal{F}^W$ yields the output instance $\mathcal{F}_P^W$, then the equality $\textsc{opt}(\mathcal{F}^W) = \textsc{opt}(\mathcal{F}_P^W)$ is guaranteed to hold. Our preprocessing is *certified*, meaning that we provide a machine-verifiable proof justifying this claimed equality. Our discussion below focuses on input instances that have solutions, but our techniques also handle the—arguably less interesting—case of $\mathcal{F}^W$ not having solutions; details are in Appendix A.5.

An overview of the workflow of our certifying MaxSAT preprocessor is shown in Figure 1. Given a WCNF instance $\mathcal{F}^W$ as input, the preprocessor proceeds in five stages (illustrated on the left in Figure 1), and then outputs a preprocessed MaxSAT instance $\mathcal{F}_P^W$ together with a pseudo-Boolean proof that $\textsc{opt}(\textsc{ObjMaxSAT}(\mathcal{F}^W)) = \textsc{opt}(\textsc{ObjMaxSAT}(\mathcal{F}_P^W))$. For certified MaxSAT preprocessing, this proof can then be fed to a formally verified checker as in Section 4 to verify that (a) the initial core constraints in the proof correspond exactly to the clauses in $\textsc{ObjMaxSAT}(\mathcal{F}^W)$, (b) each step in the proof is valid, and (c) the final core constraints in the proof correspond exactly to the clauses in $\textsc{ObjMaxSAT}(\mathcal{F}_P^W)$. Below, we provide more details on the five stages of the preprocessing flow.

Stage 1: Initialization.

An input WCNF instance $\mathcal{F}^W$ is transformed to pseudo-Boolean format by converting it to an objective-centric representation $(F^0, O^0) = \textsc{ObjMaxSAT}(\mathcal{F}^W)$ and then representing all clauses in F^0 as pseudo-Boolean constraints as described in Section 2.2. The VeriPB proof starts out with core constraints $\mathrm{PB}(F^0)$ and

	preprocessing (MaxSAT)	*proof (pseudo-Boolean)*
1. Initialization	$(\mathcal{F}^W, 0)$	$(\mathrm{PB}(F^0), O^0)$ where $(F^0, O^0) = \textsc{ObjMaxSAT}(\mathcal{F}^W)$
2. Preprocessing on WCNF	$(\mathcal{F}_1^W, \textsc{lb}^1)$	(C^1, O^1)
3. Conversion to objective-centric	$(F^2, O^2 + \textsc{lb}^1)$ where $(F^2, O^2) = \textsc{ObjMaxSAT}(\mathcal{F}_1^W)$	$(\mathrm{PB}(F^2), O^2 + \textsc{lb}^1)$
4. Preprocessing on objective-centric	(F^3, O^3)	$(\mathrm{PB}(F^3), O^3)$
5. Constant removal	(F^4, O^4) where $F^4 = F^3 \wedge (b^{W^3})$ $O^4 = O^3 - W^3 + W^3 b^{W^3}$	$(\mathrm{PB}(F^4), O^4)$
Output	Preprocessed WCNF $\mathcal{F}_P^W = (F^4, F_S^P)$	Proof of equioptimality of $\mathrm{PB}(F^0)$ under O^0 and $\mathrm{PB}(F^4)$ under O^4

Figure 1: *Overview of the five stages of certified MaxSAT preprocessing of a WCNF instance $\mathcal{F}^W$. The middle column contains the state of the working MaxSAT instance as a WCNF instance and a lower bound on its optimum cost (Stages 1–2), or as an objective-centric instance (Stages 3–5). The right column contains a tuple (C, O) with the set C of core constraints, and objective O, respectively, of the proof after each stage.*

objective O^0. The preprocessor maintains a lower bound on the optimal cost of the working instance, which is initialized to 0 for the input $\mathcal{F}^W$.

Stage 2: Preprocessing on the Initial WCNF Representation.

During preprocessing on the WCNF representation, a (very limited) set of simplification techniques are applied on the working formula. At this stage the preprocessor removes duplicate, tautological, and blocked clauses [JBH10]. Additionally, hard unit clauses are unit propagated and clauses subsumed by hard clauses are removed. Importantly, the preprocessor is performing these simplifications on a WCNF MaxSAT instance where it deals with hard and soft clauses. As the pseudo-Boolean proof has no concept of hard or soft clauses, the reformulation steps must be expressed in terms of the constraints in the proof. The next example illustrates how reasoning with different types of clauses is logged in the proof.

Example 1. Suppose the working instance has two duplicate clauses C and D. If both are hard, then the proof has two identical constraints $\mathrm{PB}(C)$ and $\mathrm{PB}(D)$ in the core set, and $\mathrm{PB}(D)$ can be deleted since it follows from $\mathrm{PB}(C)$ by reverse unit propagation (RUP). If D is instead a non-unit soft clause, the proof has the constraint $\mathrm{PB}(D \vee b_D)$ and the term $w^D b_D$ in the objective, where b_D does not

appear in any other constraint. Then in the proof we (1) remove the RUP constraint $\mathrm{PB}(D \vee b_D)$, (2) introduce $\overline{b}_D \geq 1$ by redundance-based strengthening using the witness $\{b_D \to 0\}$, (3) remove the term $w^D b_D$ from the objective, and (4) delete $\overline{b}_D \geq 1$ with the witness $\{b_D \to 0\}$.

Stage 3: Conversion to Objective-Centric Representation.

In order to apply more simplification rules in a cost-preserving way, the working instance $\mathcal{F}_1^W = (F_H^1, F_S^1)$ at the end of Stage 2 is converted into the corresponding objective-centric representation that takes the lower-bound LB inferred during Stage 1 into account. More specifically, the preprocessor next converts its working MaxSAT instance into the objective-centric instance $\mathcal{F}_2 = (F^2, O^2 + \textsc{lb})$ where $(F^2, O^2) = \textsc{ObjMaxSAT}(\mathcal{F}_1^W)$.

Here it is important to note that at the end of Stage 2, the core constraints $\mathcal{C}^1$ and objective O^1 of the proof are not necessarily $\mathrm{PB}(F^2)$ and $O^2 + \textsc{lb}$, respectively. Specifically, consider a unit soft clause $(\overline{\ell})$ of $\mathcal{F}_1^W$ obtained by shrinking a non-unit soft clause $C \supseteq (\overline{\ell})$ of the input instance, with weight w^C. Then the objective function O^2 in the preprocessor will include the term $w^C \ell$ that does not appear in the objective function O^1 in the proof. Instead, O^1 contains the term $w^C b_C$ and $\mathcal{C}^1$ the constraint $\overline{\ell} + b_C \geq 1$ where b_C is the fresh variable added to C in Stage 1. In order to "sync up" the working instance and the proof we (1) introduce $\ell + \overline{b}_C \geq 1$ to the proof with the witness $\{b_C \to 0\}$, (2) update O^1 by adding $w^C \ell - w^C b_C$, (3) remove the constraint $\ell + \overline{b}_C \geq 1$ with the witness $\{b_C \to 0\}$, and (4) remove the constraint $\overline{\ell} + b_C \geq 1$ with witness $\{b_C \to 1\}$. The same steps are logged for all soft unit clauses of $\mathcal{F}_1^W$ obtained during Stage 2. In the following stages, the preprocessor will operate on an objective-centric MaxSAT instance whose clauses correspond exactly to the core constraints of the proof.

Stage 4: Preprocessing on the Objective-Centric Representation.

During preprocessing on the objective-centric representation, more simplification techniques are applied to the working objective-centric instance and logged to the proof. We implemented proof logging for a wide range of preprocessing techniques. These include MaxSAT versions of rules commonly used in SAT solving like bounded variable elimination (BVE) [EB05, SP04], bounded variable addition [MHB13], blocked clause elimination [JBH10], subsumption elimination, self-subsuming resolution [EB05, OGMS02], failed literal elimination [Fre95, LB01, ZM88], and equivalent literal substitution [Bra04, Li00, VG05]. We also cover MaxSAT-specific preprocessing rules like TrimMaxSAT [PRB21], (group)-subsumed literal (or label) elimination (SLE) [BSJ16, KBSJ17], intrinsic at-most-ones [IMM19, IBJ22], binary core removal (BCR) [Gim64, KBSJ17], label matching [KBSJ17], and hardening [ABGL12, IBJ22, MHM12]. Here we give examples for BVE, SLE, label matching, and BCR—the rest are detailed in Appendix A. In the following descriptions, let (F, O) be the current objective-centric working instance.

Bounded Variable Elimination (BVE) [EB05, SP04]. BVE eliminates from F a variable x that does not appear in the objective by replacing all clauses in which either x or $\overline{x}$ appears with the non-tautological clauses in $\{C \vee D \mid C \vee x \in F, D \vee \overline{x} \in F\}$.

An application of BVE is logged as follows: (1) each non-tautological constraint PB($C \vee D$) is added by summing the existing constraints PB($C \vee x$) and PB($D \vee \overline{x}$) and saturating, after which (2) each constraint of the form PB($C \vee x$) and PB($D \vee \overline{x}$) is deleted with the witness $x \to 1$ or $x \to 0$, respectively.

Label Matching [KBSJ17]. Label matching allows merging pairs of objective variables that can be deduced to not both be set to 1 by optimal solutions. Assume that (i) F contains the clauses $C \vee b_C$ and $D \vee b_D$, (ii) b_C and b_D are objective variables with the same coefficient w in O, and (iii) $C \vee D$ is a tautology. Then label matching replaces b_C and b_D with a fresh variable b_{CD}, i.e., replaces $C \vee b_C$ and $D \vee b_D$ with $C \vee b_{CD}$ and $D \vee b_{CD}$ and adds $-wb_C - wb_D + wb_{CD}$ to O.

As $C \vee D$ is a tautology there is some literal ℓ such that $\overline{\ell} \in C$ and $\ell \in D$. Label matching is logged via the following steps: (1) introduce the constraint $\overline{b}_C + \overline{b}_D \geq 1$ with the witness $\{b_C \to \ell, b_D \to \overline{\ell}\}$, (2) introduce the constraints $b_{CD} + \overline{b}_C + \overline{b}_D \geq 2$ and $\overline{b}_{CD} + b_C + b_D \geq 1$ by redundance; these correspond to $b_{CD} = b_C + b_D$ which holds even though the variables are binary due to the constraint added in the first step, (3) update the objective by adding $-wb_C - wb_D + wb_{CD}$ to it, (4) introduce the constraints PB($C \vee b_{CD}$) and PB($D \vee b_{CD}$) which are RUP, (5) delete PB($C \vee b_C$) and PB($D \vee b_D$) with the witness $\{b_C \to \overline{\ell}, b_D \to \ell\}$, (6) delete the constraint $b_{CD} + \overline{b}_C + \overline{b}_D \geq 2$ with the witness $\{b_C \to 0, b_D \to 0\}$ and $\overline{b}_{CD} + b_C + b_D \geq 1$ with the witness $\{b_C \to 1, b_D \to 0\}$, (7) delete $\overline{b}_C + \overline{b}_D \geq 1$ with the witness $\{b_C \to 0\}$.

Subsumed Literal Elimination (SLE) [BSJ16, IBJ22]. Given two non-objective variables x and y such that (i) $\{C \mid C \in F, y \in C\} \subseteq \{C \mid C \in F, x \in C\}$ and (ii) $\{C \mid C \in F, \overline{x} \in C\} \subseteq \{C \mid C \in F, \overline{y} \in C\}$, subsumed literal elimination (SLE) allows fixing $x = 1$ and $y = 0$. This is proven by (1) introducing $x \geq 1$ and $\overline{y} \geq 1$, both with witness $\{x \to 1, y \to 0\}$, (2) simplifying the constraint database via propagation, and (3) deleting the constraints introduced in the first step as neither x nor y appears in any other constraints after simplification.

If x and y are objective variables, the application of SLE additionally requires that: (iii) the coefficient in the objective of x is at most as high as the coefficient of y. Then the value of x is not fixed as it would incur cost. Instead, only $y = 0$ is fixed and y removed from the objective. Intuitively, conditions (i) and (ii) establish that the values of x and y can always be flipped to 0 and 1, respectively, without falsifying any clauses. If neither of the variables is in the objective, this flip does not increase the cost of any solutions. Otherwise, condition (iii) ensures that the flip does not make the solution worse, i.e., increase its cost.

Binary Core Removal (BCR) [Gim64, KBSJ17]. Assume that the following four prerequisites hold: (i) F contains a clause $b_C \vee b_D$ for two objective variables b_C and b_D, (ii) b_C and b_D have the same coefficient w in O, (iii) the negations b_C and b_D do not appear in any clause of F, and (iv) both b_C and b_D appear in at

least one other clause of F but not together in any other clause of F. Binary core removal replaces all clauses containing b_C or b_D with the non-tautological clauses in $\{C \vee D \vee b_{CD} \mid C \vee b_C \in F, D \vee b_D \in F\}$, where b_{CD} is a fresh variable, and modifies the objective function by adding $-wb_C - wb_D + wb_{CD} + w$ to it.

BCR is logged as a combination of the so-called *intrinsic at-most-ones* technique [IMM19, IBJ22] and BVE. Applying intrinsic at most ones on the variables b_C and b_D introduces a new clause $(\overline{b}_C \vee \overline{b}_D \vee b_{CD})$ and adds $-wb_C - wb_D + wb_{CD} + w$ to the objective. Our proof for intrinsic at most ones is the same as the one presented in [BBN+23]. As this step removes b_C and b_D from the objective, both can now be eliminated via BVE.

Stage 5: Constant Removal and Output.

After objective-centric preprocessing, the final objective-centric instance (F^3, O^3) is converted back to a WCNF instance. Before doing so, the constant term W_3 of O^3 is removed by introducing a fresh variable b^{W_3}, and setting $F^4 = F^3 \wedge (b^{W_3})$ and $O^4 = O^3 - W_3 + W_3 b^{W_3}$. This step is straightforward to prove.

Finally, the preprocessor outputs the WCNF instance $\mathcal{F}_P^W = (F^4, F_S^P)$ that has F^4 as hard clauses. the set F_S^P of soft clauses consists of a unit soft clause $(\overline{\ell})$ of weight c for each term $c \cdot \ell$ in O^4. The preprocessor also outputs the final proof of the fact that the minimum-cost of solutions to the pseudo-Boolean formula PB(F^0) under O^0 is the same as that of PB(F^4) under O^4, i.e. that $\textsc{opt}(\textsc{ObjMaxSAT}(\mathcal{F}^W)) = \textsc{opt}(\textsc{ObjMaxSAT}(\mathcal{F}_P^W))$.

3.2 Worked Example of Certified Preprocessing

We give a worked-out example of certified preprocessing of the instance $\mathcal{F}^W = (F_H, F_S)$ where $F_H = \{(x_1 \vee x_2), (\overline{x}_2)\}$ and three soft clauses: $(\overline{x}_1)$ with weight 1, $(x_3 \vee \overline{x}_4)$ with weight 2, and $(x_4 \vee \overline{x}_5)$ with weight 3. The proof for one possible execution of the preprocessor on this input instance is detailed in Table 1.

During Stage 1 (Steps 1–4 in Table 1), the core constraints of the proof are initialized to contain the four constraints corresponding to the hard and non-unit soft clauses of $\mathcal{F}^W$ (IDs (1)–(4) in Table 1), and the objective to $x_1 + 2b_1 + 3b_2$, where b_1 and b_2 are fresh variables added to the non-unit soft clauses of $\mathcal{F}^W$.

During Stage 2 (Steps 5–9), the preprocessor fixes $x_2 = 0$ via unit propagation by removing x_2 from the clause $(x_1 \vee x_2)$, and then removing the unit clause $(\overline{x}_2)$. The justification for fixing $x_2 = 0$ are Steps 5–7. Next the preprocessor fixes $x_1 = 1$ which (i) removes the hard clause (x_1), and (ii) increases the lower bound on the optimal cost by 1. The justification for fixing $x_1 = 1$ are Steps 8 and 9 of Table 1. At this point—at the end of Stage 2—the working instance $\mathcal{F}_1^W = (F_H^1, F_S^1)$ has $F_H^1 = \{\}$ and $F_S^1 = \{(x_3 \vee \overline{x}_4), (x_4 \vee \overline{x}_5)\}$.

In Stage 3, the preprocessor converts its working instance into the objective-centric representation (F, O) where $F = \{(x_3 \vee \overline{x}_4 \vee b_1), (x_4 \vee \overline{x}_5 \vee b_2)\}$ and $O = 2b_1 + 3b_2 + 1$, which exactly matches the core constraints and objective of the proof after Step 9. Thus, in this instance, the conversion does not result in any proof logging steps. Afterwards, during Stage 4 (Steps 10–17), the preprocessor applies

Table 1: *Example proof produced by a certifying preprocessor. The column (ID) refers to constraint IDs in the pseudo-Boolean proof. The column (Step) indexes all proof logging steps and is used when referring to the steps in the discussion. The letter ω is used for the witness substitution in redundance-based strengthening steps.*

Step	ID	Type	Justification	Objective
1	(1)	add $x_1 + x_2 \geq 1$	input	$x_1 + 2b_1 + 3b_2$
2	(2)	add $\overline{x}_2 \geq 1$	input	$x_1 + 2b_1 + 3b_2$
3	(3)	add $x_3 + \overline{x}_4 + b_1 \geq 1$	input	$x_1 + 2b_1 + 3b_2$
4	(4)	add $x_4 + \overline{x}_5 + b_2 \geq 1$	input	$x_1 + 2b_1 + 3b_2$
	Unit propagation: fix $x_2 = 0$, constraint (2)			
5	(5)	add $x_1 \geq 1$	(1) + (2)	$x_1 + 2b_1 + 3b_2$
6		delete (1)	RUP	$x_1 + 2b_1 + 3b_2$
7		delete (2)	$\omega\colon \{x_2 \to 0\}$	$x_1 + 2b_1 + 3b_2$
	Unit propagation; fix $x_1 = 1$, constraint (5)			
8		add $-x_1 + 1$ to obj.	(5)	$2b_1 + 3b_2 + 1$
9		delete (5)	$\omega\colon \{x_1 \to 1\}$	$2b_1 + 3b_2 + 1$
	BVE: eliminate x_4			
10	(6)	add $x_3 + b_1 + \overline{x}_5 + b_2 \geq 1$	(3) + (4)	$2b_1 + 3b_2 + 1$
11		delete (3)	$\omega\colon \{x_4 \to 0\}$	$2b_1 + 3b_2 + 1$
12		delete (4)	$\omega\colon \{x_4 \to 1\}$	$2b_1 + 3b_2 + 1$
	Subsumed literal elimination: $\overline{b}_2$			
13	(7)	add $\overline{b}_2 \geq 1$	$\omega\colon \{b_2 \to 0, b_1 \to 1\}$	$2b_1 + 3b_2 + 1$
14		add $-3b_2$ to obj.	(7)	$2b_1 + 1$
15	(8)	add $x_3 + b_1 + \overline{x}_5 \geq 1$	(6) + (7)	$2b_1 + 1$
16		delete (6)	RUP	$2b_1 + 1$
17		delete (7)	$\omega\colon \{b_2 \to 0\}$	$2b_1 + 1$
	Remove objective constant			
18	(9)	add $b_3 \geq 1$	$\omega\colon \{b_3 \to 1\}$	$2b_1 + 1$
19		add $b_3 - 1$ to obj.	(9)	$2b_1 + b_3$

BVE in order to eliminate x_4 (Steps 10–12) and SLE to fix b_2 to 0 (Steps 13–17). Finally, Steps 18 and 19 represent Stage 5, i.e., the removal of the constant 1 from the objective. After these steps, the preprocessor outputs the preprocessed instance $\mathcal{F}_P^W = (F_H^P, F_S^P)$, where $F_H^P = \{(x_3 \vee \overline{x}_5 \vee b_1), (b_3)\}$ and F_S^P contains two clauses: $(\overline{b}_1)$ with weight 2, and $(\overline{b}_3)$ with weight 1.

4 Verified Proof Checking for Preprocessing Proofs

This section presents our new workflow for formally verified, end-to-end proof checking of MaxSAT preprocessing proofs based on pseudo-Boolean reasoning; an overview of this workflow is shown in Figure 2. To realize this workflow, we extended the VeriPB tool and its proof format to support a new *output section* for declaring (and checking) reformulation guarantees between input and output

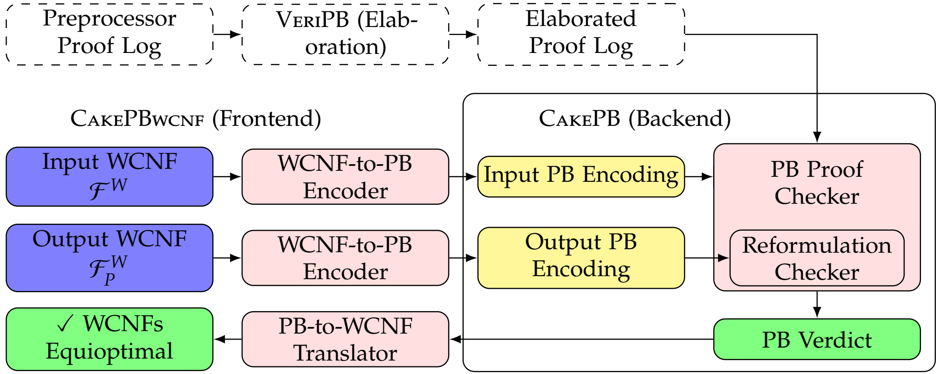


Figure 2: *Workflow for end-to-end verified MaxSAT preprocessing proof checking.*

PBO instances (Section 4.1); we similarly modified CakePB [GMM+24] a verified proof checker to support the updated proof format (Section 4.2); finally, we built a verified frontend, CakePBwcnf, which mediates between MaxSAT WCNF instances and PBO instances (Section 4.3). Our formalization is carried out in the HOL4 proof assistant [SN08] using CakeML tools [GMKN17, MO14, TMK+19] to obtain a verified executable implementation of CakePBwcnf.

In the workflow in Figure 2, the MaxSAT preprocessor produces a reformulated output WCNF together with a proof of equioptimality with the input WCNF. This proof is elaborated by VeriPB and then checked by CakePBwcnf, resulting in a verified *verdict*—in case of success, the input and output WCNFs are equioptimal. This workflow also supports verified checking of WCNF MaxSAT solving proofs (where the output parts of the flow are omitted).

4.1 Output Section for Pseudo-Boolean Proofs

Given an input PBO instance (F, O), the VeriPB proof system as described in Section 2.1 maintains the invariant that the core constraints C (and current objective) are equioptimal to the input instance. Utilizing this invariant, the new *output section* for VeriPB proofs allows users to optionally specify an output PBO instance (F', O') at the end of a proof. This output instance is claimed to be a reformulation of the input which is either: (i) *derivable*, i.e., satisfiability of F implies satisfiability of F', (ii) *equisatisfiable* to F, or (iii) *equioptimal* to (F, O). These are increasingly stronger claims about the relationship between the input and output instances. After checking a pseudo-Boolean derivation, VeriPB runs reformulation checking which, e.g., for equioptimality, checks that $C \subseteq F'$, $F' \subseteq C$, and that the respective objective functions are syntactically equal after normalization; other reformulation guarantees are checked analogously.

The VeriPB tool supports an *elaboration* mode [GMM+24], where in addition to checking the proof it also converts it from *augmented format* to *kernel format*. The augmented format contains syntactic sugar rules to facilitate proof logging for solvers and preprocessors like MaxPre, while the kernel format is supported by

the formally verified proof checker CakePB. The new output section is passed unchanged from augmented to kernel format during elaboration.

4.2 Verified Proof Checking for Reformulations

There are two main verification tasks involved in extending CakePB with support for the output section. The first task is to verify soundness of all cases of reformulation checking. Formally, the equioptimality of an input PBO instance *fml*, *obj* and its output counterpart *fml*′, *obj*′ is specified as follows:

$$\begin{aligned}&\mathsf{sem_output}\ \mathit{fml}\ \mathit{obj}\ \mathsf{None}\ \mathit{fml}'\ \mathit{obj}'\ \mathsf{Equioptimal} \stackrel{\text{def}}{=}\\ &\forall v.\ (\exists w.\ \mathsf{satisfies}\ w\ \mathit{fml} \land \mathsf{eval_obj}\ \mathit{obj}\ w \le v) \iff\\ &\quad(\exists w'.\ \mathsf{satisfies}\ w'\ \mathit{fml}' \land \mathsf{eval_obj}\ \mathit{obj}'\ w' \le v)\end{aligned}$$

This definition says that, for all values v, the input instance has a satisfying assignment with objective value less than or equal to v iff the output instance also has such an assignment; note that this implies (as a special case) that *fml* is satisfiable iff *fml*′ is satisfiable. The verified correctness theorem for CakePB says that *if* CakePB successfully checks a pseudo-Boolean proof in kernel format and prints a verdict declaring equioptimality, then the input and output instances are indeed equioptimal as defined in sem_output.

The second task is to develop verified optimizations to speedup proof steps which occur frequently in preprocessing proofs; some code hotspots were also identified by profiling the proof checker against proofs generated by MaxPre. Similar (unverified) versions of these optimizations are also used in VeriPB. These optimizations turned out to be necessary in practice—they mostly target steps which, when naïvely implemented, have quadratic (or worse) time complexity in the size of the constraint database.

Optimizing Reformulation Checking. The most expensive step in reformulation checking for the output section is to ensure that the core constraints $\mathcal{C}$ are included in the output formula and vice versa (possibly with permutations and duplicity). Here, CakePB normalizes all pseudo-Boolean constraints involved to a canonical form and then copies both $\mathcal{C}$ and the output formula into respective array-backed hash tables for fast membership tests.

Optimizing Redundance and Checked Deletion Rules. A naïve implementation of these two rules would require iterating over the entire constraints database when checking all subproofs in (1) for the right-hand-side constraints $(\mathcal{C} \cup \mathcal{D} \cup \{C\})\restriction_\omega$ $\cup\{O \ge O\restriction_\omega\}$. An important observation here is that preprocessing proofs frequently use substitutions ω that only involve a small number of variables (often a single variable, which in addition is fresh in the important special case of *reification* constraints $z \Leftrightarrow C$ encoding that z is true precisely when the constraint C is satisfied). Consequently, most of the constraints $(\mathcal{C} \cup \mathcal{D} \cup \{C\})\restriction_\omega$ can be skipped when checking redundance because they are unchanged by the substitution. Similarly, the constraint $O \ge O\restriction_\omega$ is expensive to construct when the objective O contains many terms, but this construction can be skipped if no variables being

$$\mathsf{sat_hard}\ w\ \mathit{wfml} \stackrel{\text{def}}{=} \forall C.\ \mathsf{mem}\ (0,C)\ \mathit{wfml} \Rightarrow \mathsf{sat_clause}\ w\ C$$
$$\mathsf{weight_clause}\ w\ (n,C) \stackrel{\text{def}}{=} \mathsf{if\ sat_clause}\ w\ C\ \mathsf{then}\ 0\ \mathsf{else}\ n$$
$$\mathsf{cost}\ w\ \mathit{wfml} \stackrel{\text{def}}{=} \mathsf{sum}\ (\mathsf{map}\ (\mathsf{weight_clause}\ w)\ \mathit{wfml})$$
$$\mathsf{opt_cost}\ \mathit{wfml} \stackrel{\text{def}}{=} \mathsf{if}\ \neg\exists w.\ \mathsf{sat_hard}\ w\ \mathit{wfml}\ \mathsf{then\ None}\ \mathsf{else\ Some}\ (\mathsf{min}_{\mathsf{set}}\ \{\ \mathsf{cost}\ w\ \mathit{wfml} \mid \mathsf{sat_hard}\ w\ \mathit{wfml}\ \}\ )$$

Figure 3: *Formalized semantics for MaxSAT WCNF problems.*

substituted occur in O. CakePB stores a lazily-updated mapping of variables to their occurrences in the constraint database and the objective, which it uses to detect these cases.

The occurrence mapping just discussed is crucial for performance due to the frequency of steps involving witnesses for preprocessing proofs, but incurs some memory overhead in the checker. More precisely, every variable occurrence in any constraint in the database corresponds to exactly one ID in the mapping. Thus, the overhead of storing the mapping is in the worst case quadratic in the number of constraints, but it is still linear in the total space usage for the constraints database.

4.3 Verified WCNF Frontend

The CakePBwcnf frontend mediates between MaxSAT WCNF problems and pseudo-Boolean optimization problems native to CakePB. Accordingly, the correctness of CakePBwcnf is stated in terms of MaxSAT semantics, i.e., the encoding, underlying pseudo-Boolean semantics, and proof system are all formally verified. In order to trust CakePBwcnf, one *only* has to carefully inspect the formal definition of MaxSAT semantics shown in Figure 3 to make sure that it matches the informal definition in Section 2.2. Here, each clause C is paired with a natural number n, where $n = 0$ indicates a hard clause and when $n > 0$ it is the weight of C. The optimal cost of a weighted CNF formula *wfml* is None (representing ∞) if no satisfying assignment to the hard clauses exist; otherwise, it is the minimum cost among all satisfying assignments to the hard clauses.

There and Back Again. CakePBwcnf contains a verified WCNF-to-PB encoder implementing the encoding described in Section 2.2. Its correctness theorems are shown in Figure 4, where the two lemmas in the top row relate the satisfiability and cost of the WCNF to its PB optimization counterpart after running wcnf_to_pbf (and vice versa), see Observation 1. Using these lemmas, the final theorem (bottom row) shows that equioptimality for two (encoded) PB optimization problems can be *translated* back to equioptimality for the input and preprocessed WCNFs.

Putting Everything Together. The final verification step is to specialize the end-to-end machine code correctness theorem for CakePB to the new frontend. The resulting theorem for CakePBwcnf is shown abridged in Figure 5; a detailed explanation of similar CakeML-based theorems is available elsewhere [GMM+24, THM23] so we do not go into details here. Briefly, the theorem says that whenever

⊢ wfml_to_pbf $wfml$ = (obj,pbf) ∧
  satisfies w (set pbf) ⇒
   ∃w'. sat_hard w' $wfml$ ∧
      cost w' $wfml$ ≤ eval_obj obj w

⊢ wfml_to_pbf $wfml$ = (obj,pbf) ∧
  sat_hard w $wfml$ ⇒
   ∃w'. satisfies w' (set pbf) ∧
      eval_obj obj w' = cost w $wfml$

⊢ full_encode $wfml$ = (obj,pbf) ∧ full_encode $wfml'$ = (obj',pbf') ∧
  sem_output (set pbf) obj None (set pbf') obj' Equioptimal ⇒
   opt_cost $wfml$ = opt_cost $wfml'$

Figure 4: *Correctness theorems for the WCNF-to-PB encoding.*

⊢ cake_pb_wcnf_run cl fs mc ms ⇒
  ∃out err.
   extract_fs fs (cake_pb_wcnf_io_events cl fs) =
    Some (add_stdout (add_stderr fs err) out) ∧
   (length cl = 4 ∧ isSuffix "s `VERIFIED OUTPUT EQUIOPTIMAL\n`" out ⇒
    ∃$wfml$ $wfml'$.
     get_fml fs (el 1 cl) = Some $wfml$ ∧ get_fml fs (el 3 cl) = Some $wfml'$ ∧
     opt_cost $wfml$ = opt_cost $wfml'$)

Figure 5: *Abridged final correctness theorem for CakePBwcnf.*

the verdict string "s `VERIFIED OUTPUT EQUIOPTIMAL`" is printed (as a suffix) to the standard output by an execution of CakePBwcnf, then the two input files given on the command line parsed to equioptimal MaxSAT WCNF instances.

5 Experiments

We upgraded the MaxSAT preprocessor MaxPre 2.1 [IBJ22, JBIJ23, KBSJ17] to MaxPre 2.2, which produces proof logs in the VeriPB format [BMM+23]. MaxPre 2.2 is available at the MaxPre 2 repository [Maxa]. The generated proofs were elaborated using VeriPB [Ver] and then checked by the verified proof checker CakePBwcnf. As benchmarks we used the 558 weighted and 572 unweighted MaxSAT instances from the MaxSAT Evaluation 2023 [Max23].

The experiments were conducted on 11th Gen Intel(R) Core(TM) i5-1145G7 @ 2.60GHz CPUs with 16 GB of memory, a solid state drive as storage, and Rocky Linux 8.5 as operating system. Each benchmark ran exclusively on a node and the memory was limited to 14 GB. The time for MaxPre was limited to 300 seconds. There is an option to let MaxPre know about this time limit, but we did not use this option since MaxPre then decides which techniques to try based on how much time remains. This would have made it very hard to get reliable measurements of the overhead when proof logging is switched on in the preprocessor. The time limits for both VeriPB and CakePBwcnf were set to 6000 seconds to get as many instances checked as possible.

The main focus of our evaluation was the default setting of MaxPre, which does not use some of the techniques mentioned in Section 3 (or Appendix A). We

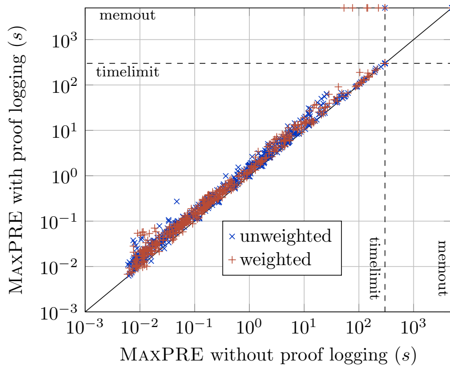


Figure 6: *Proof logging overhead for MaxPre.*

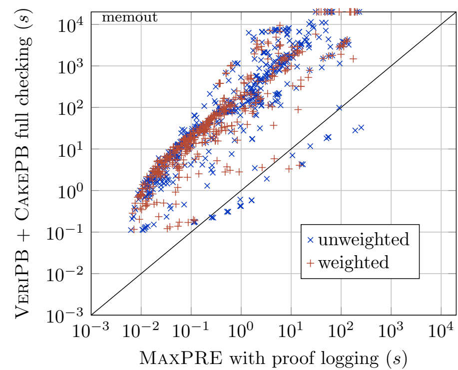


Figure 7: *MaxPre vs. combined proof checking running time.*

also conducted experiments with all techniques enabled to check the correctness of the proof logging implementation for all preprocessing techniques. The data and source code from our experiments can be found in [IOT+24].

The goal of the experiments was to answer the following questions:

RQ1. How much extra time is required to write the proof for the preprocessor?

RQ2. How long does proof checking take compared to proof generation?

To answer the first question, in Figure 6 we compare MaxPre with and without proof logging. In total, 1081 instances were successfully preprocessed by MaxPre without proof logging. With proof logging enabled, 8 fewer instances were preprocessed due to either time- or memory-outs. For the successfully preprocessed instances, the geometric mean of the proof logging overhead is 46% of the running time, and 95% of the instances were preprocessed with proof logging in at most twice the time required without proof logging.

Our comparison between proof generation and proof checking is based on the 1073 instances for which preprocessing with proof logging was successful. Out of these, 1021 instances were successfully checked and elaborated by VeriPB. For 991 instances the verdicts were confirmed by the formally verified proof checker CakePBwcnf, with the remaining instances being time- or memory-outs. This shows the practical viability of our approach, as the vast majority of preprocessing proofs were checked within the resource limits.

A scatter plot comparing the running time of MaxPre with proof logging enabled against the combined checking process is shown in Figure 7. For the combined checking time, we only consider the instances that have been successfully checked by CakePBwcnf. In the geometric mean, the time for the combined verified checking pipeline of VeriPB elaboration followed by CakePBwcnf checking is 113× the preprocessing time of MaxPre. A general reason for this overhead is that the preprocessor has more MaxSAT application-specific context than the pseudo-Boolean checker, so the preprocessor can log proof steps without performing the actual reasoning while the checker must ensure that those steps are sound in an

application-agnostic way. An example for this is reification: as the preprocessor knows its reification variables are fresh, it can easily emit redundance steps that witness on those variables; but the checker has to verify freshness against its own database. Similar behaviour has been observed in other applications of pseudo-Boolean proof logging [GMNO22, HOGN24].

To analyse further the causes of proof checking overhead, we also compared VeriPB to CakePBwcnf. The checking of the elaborated kernel proof with CakePBwcnf is 6.7× faster than checking and elaborating the augmented proof with VeriPB. This suggests that the bottleneck for proof checking is VeriPB; VeriPB *without* elaboration is about 5.3× slower than CakePBwcnf. As elaboration is a necessary step before running CakePBwcnf, improving the performance of VeriPB would benefit the performance of the pipeline as a whole. One specific feature that seems desirable would be to augment RUP rule applications with LRAT-style hints [CHH+17], so that VeriPB would not need to perform unit propagation to elaborate RUP steps to cutting planes derivations. Though these types of engineering challenges are important to address, they are beyond the scope of the current paper and we have to leave them as future work.

6 Conclusion

In this work, we show how to use pseudo-Boolean proof logging to certify correctness of the MaxSAT preprocessing phase, extending previous work for the main solving phase in unweighted model-improving solvers [VDB22] and general core-guided solvers [BBN+23]. As a further strengthening of previous work, we present a fully formally verified toolchain which provides end-to-end verification of correctness.

In contrast to SAT solving, there is a rich variety of techniques in maximum satisfiability solving, and it still remains to design pseudo-Boolean proof logging methods for general, weighted, model-improving MaxSAT solvers [ES06, LP10, PRB18] and *implicit hitting set (IHS)* MaxSAT solvers [DB11, DB13] with *abstract cores* [BBP20]. Nevertheless, our work adds further weight to the conclusion that pseudo-Boolean reasoning seems like a very promising foundation for MaxSAT proof logging. We are optimistic that this work is another step on the path towards general adoption of proof logging in the context of SAT-based optimization.

Acknowledgements

This work has been financially supported by the University of Helsinki Doctoral Programme in Computer Science DoCS, the Research Council of Finland under grants 342145 and 346056, the Swedish Research Council grants 2016-00782 and 2021-05165, the Independent Research Fund Denmark grant 9040-00389B, the Wallenberg AI, Autonomous Systems and Software Program (WASP) funded by the Knut and Alice Wallenberg Foundation, and by A*STAR, Singapore. Part of this work was carried out while some of the authors participated in the extended reunion of the semester program *Satisfiability: Theory, Practice, and Beyond* in the spring of 2023 at the Simons Institute for the Theory of Computing at UC Berkeley.

We also acknowledge useful discussions at the Dagstuhl workshops 22411 *Theory and Practice of SAT and Combinatorial Solving* and 23261 *SAT Encodings and Beyond*. The computational experiments were enabled by resources provided by LUNARC at Lund University.

Appendix A Complete Overview of Proof Logging for MaxSAT Preprocessing

In this appendix, we provide a complete overview of proof logging for the preprocessing techniques implemented by MaxPre. As we already presented proof logging for bounded variable elimination, subsumed literal elimination, label matching and binary core removal in Section 3 of the paper, we do not present those techniques here. In addition, we do not include intrinsic at-most-ones (even though implemented in MaxPre), as it is already discussed in [BBN+23].

A.1 Fixing Variables

Many of the preprocessing techniques can fix variables (or literals) to either 0 or 1. We describe here the generic procedure that is invoked when a variable is fixed. Assume that a preprocessing technique decides to fix $\ell = 1$ for a literal ℓ. Then, in the preprocessor, each clause $C \lor \overline{\ell}$ is replaced by clause C, i.e., falsified literal $\overline{\ell}$ is removed. Additionally, each clause $C \lor \ell$ is removed (as they are satisfied when $\ell = 1$).

In the proof, we do the following. First, the technique that fixes $\ell = 1$, ensures that constraint $\ell \geq 1$ is in the core constraints of the proof. It may be that $\ell \geq 1$ is already in the core constraints of the proof (i.e. instance has a unit clause (ℓ)), or it may be that $\ell \geq 1$ needs to be introduced as a new constraint. The details on how $\ell \geq 1$ is introduced depends on the specific technique that is fixing $\ell = 1$. Now, assuming $\ell \geq 1$ is in the core constraints, the following procedure is invoked.

(1) If ℓ or $\overline{\ell}$ appears in the objective function, the objective function is updated.

(2) For each clause $C \lor \overline{\ell}$, the constraint $\mathrm{PB}(C)$ is introduced as a sum of $\mathrm{PB}(C \lor \overline{\ell})$ and $\ell \geq 1$.

(3) Each constraint $\mathrm{PB}(C \lor \ell)$ is deleted (as a RUP constraint).

(4) Finally, the core constraint $\ell \geq 1$ is deleted last with witness $\{\ell \to 1\}$.

A.2 Preprocessing on the Initial WCNF Representation

We explain the preprocessing techniques that can be applied during preprocessing on the WCNF representation, detailing especially how the different types of clauses are handled. The preprocessing techniques applied on the WCNF representation only modify a clause C by either removing a literal ℓ from C or removing C entirely. With this intuition, given an input WCNF instance $\mathcal{F}^W = (F_H, F_S)$ and a working instance $\mathcal{F}_1^W = (F_H^1, F_S^1)$ each clause in $\mathcal{F}_1^W$ is one of the following three types:

(1) A hard clause $C \in F_H^1$ that is a subset or equal to a hard clause $C \subseteq C^{\text{orig}} \in F_H$ of $\mathcal{F}^W$.

(2) An *originally unit soft clause*, i.e., a soft clause $C \in F_S^1$ that is equal to a unit soft clause in F_S.

(3) An *originally non-unit soft clause*, i.e., $C \in F_S^1$ that is a subset or equal to a non-unit soft clause $C \subseteq C^{\text{orig}} \in F_S$ of $\mathcal{F}^W$.

With this we next detail how the preprocessing rules permitted on the WCNF representation are logged. In the following, we assume a fixed working WCNF instance.

Duplicate Clause Removal.

In the paper we discussed how to log the removal of two duplicate clauses C and D when: (i) both are hard, or (ii) C is hard and D is an originally non-unit soft clause. Here we detail the remaining cases.

Assume first that both C and D are originally non-unit duplicate soft clauses with weights w^C and w^D, respectively. Then the proof has the core constraints $\text{PB}(C \vee b_C)$ and $\text{PB}(D \vee b_D)$ and its objective the terms $w^C b_C$ and $w^D b_D$. The removal of D is logged as follows.

(1) Introduce the constraints: $\overline{b}_C + b_D \geq 1$ with the witness $\{b_C \to 0\}$ and $b_C + \overline{b}_D \geq 1$, with the witness $\{b_D \to 0\}$ to the core set. These encode $b_C = b_D$.

(2) Update the objective by adding $-w^C b_C + w^C b_D$ to it, conceptually increasing the coefficient of b_D by w^C.

(3) Remove the constraints introduced in step (1) using the same witnesses.

(4) Remove the (RUP) constraint $\text{PB}(D \vee b_C)$.

If $C = (\overline{\ell})$ is originally a unit soft clause but $D = (\overline{\ell})$ is originally a non-unit soft clause, then the core constraints of the proof include constraint $\text{PB}(\ell \vee b_D)$ and the objective of the proof the terms $w^C \ell$ and $w^D b_D$. The removal of D is logged similarly to the previous case with the literal b_C replaced with ℓ.

The case of two duplicate originally unit soft clauses does not require proof logging since the corresponding terms in the objective are automatically summed.

Tautology Removal.

If a clause is a tautology, it is also a RUP clause. Thus, a tautological hard clause is simply deleted. The removal of a tautological soft clause additionally requires updating the objective.

More specifically, assume C is a tautological soft clause of weight w^C. Then C is originally non-unit, so the proof has a constraint $\text{PB}(C \vee b_C)$ and its objective the term $w^C b_C$. The removal of C is logged with the following steps:

(1) Delete the (RUP) constraint $\text{PB}(C \vee b_C)$.

(2) Introduce the constraint $\overline{b}_C \geq 1$ with witness $\{b_C \to 0\}$ and move the new constraint to the core set.

(3) Update the objective by adding $-w^C b_C$ to it.

(4) Remove the constraint introduced in step (2) with the same witness.

Unit Propagation of Hard Clauses.

If the instance contains a (hard) unit clause (l), the literal l is fixed to 1 with the method of fixing variables described in Section A.1.

Removal of Empty Soft Clauses.

If the instance contains an empty soft clause C—either as input or as a consequence of e.g., unit propagation—it is removed and the lower bound increased by its weight w^C. If C was originally non-unit, the core constraints of the proof contain the constraint $b_C \geq 1$ and the objective the term $w^C b_C$. The removal of C is logged by the following steps:

(1) Update the objective by adding $-w^C b_C + w^C$.

(2) Delete the constraint $b_C \geq 1$ with the witness $\{b_C \to 1\}$.

If $C = (\ell)$ is an originally unit soft clause the objective is updated in conjunction with the literal ℓ getting fixed to 0, as described in Section A.1. Thus, no further steps are required.

Blocked Clause Elimination (BCE) [JBH10].

Our implementation of BCE considers a clause $C \vee \ell$ blocked (on the literal ℓ) if for each clause $D \vee \overline{\ell}$ there is a literal $\ell' \in D$ for which $\overline{\ell'} \in C$.

When preprocessing on the objective-centric representation, BCE considers only literals ℓ for which neither ℓ nor $\overline{\ell}$ appears in the objective function. During initial WCNF preprocessing stage, there are no requirements for literal ℓ. (Notice that whenever there is a unit clause $(\overline{\ell})$, $C \vee \ell$ is not blocked on the literal ℓ.)

The removal of a blocked clause is logged as the deletion of the corresponding constraint $\mathrm{PB}(C \vee \ell)$ with the witness $\{\ell \to 1\}$. If $C \vee \ell$ is an (originally non-unit) soft clause, the objective function is also updated exactly as with tautology removal.

Subsumption Elimination.

A clause D is subsumed by the clause C if $C \subseteq D$. Whenever the subsuming clause C is hard, D is removed as a RUP clause. If D is soft, the objective function is updated exactly as with tautology removal.

A.3 Preprocessing on Objective-Centric Representation

We detail how the preprocessing techniques that are applied on the objective-centric representation (F, O) of the working instance are logged. In addition to these, the preprocessor can also apply the techniques detailed in Section A.2.

TrimMaxSAT [PRB21].

The TrimMaxSAT technique heuristically looks for a set of literals N s.t. every solution ρ to F assigns each $\ell \in N$ to 0, or more formally, F entails the unit clause $(\overline{\ell})$. All such literals are fixed by the generic procedure (recall Section A.1). The literals to be fixed are identified by iterative calls to an (incremental) SAT solver [ES03, MLM21] under different assumptions.

In order to log the TrimMaxSAT technique we log the proof produced by each SAT solver call into the derived set of constraints in our PB proof. After the set N is identified, we make $|N|$ extra SAT calls, one for each $\ell \in N$. Each call is made assuming the value of ℓ to 1. Due to the properties of TrimMaxSAT and SAT-solvers, the result will be UNSAT, after which $\overline{\ell} \geq 1$ will be RUP w.r.t to the current set of core and derived constraints. As such it is added and moved to core in order to invoke the generic variable fixing procedure. Finally, when TrimMaxSAT will not be used any more, all constraints added to the derived set by the SAT solver are removed.

Self-Subsuming Resolution (SSR) [EB05, OGMS02].

Given clauses $C \vee l$ and $D \vee \overline{\ell}$ such that C subsumes D and ℓ is not in the objective, SSR substitutes D for $D \vee \overline{\ell}$. The proof has two steps: (1) Introduce PB(D) as a new RUP constraint. (2) Remove PB($D \vee \overline{\ell}$) as it is RUP.

Group-Subsumed Label Elimination (GSLE) [KBSJ17].

Let b be an objective variable that has the coefficient c^b in O, and L a set of objective variables such that each $b_i \in L$ has coefficient c^i in O. Assume then that (i) $c^b \geq \sum_{b_i \in L} c^i$, (ii) the negation of b or any variables in L do not appear in any clauses, and (iii) $\{C \mid b \in C\} \subseteq \{D \mid \exists b \in L : b \in D\}$. Then, an application of GSLE fixes $b = 0$. To prove an application of GSLE, we introduce the constraint $\overline{b} \geq 1$ with the witness $\{b \rightarrow 0, b_i \rightarrow 1 \mid b_i \in L\}$, and invoke the generic variable fixing procedure detailed in Section A.1 to fix $b = 0$.

Bounded Variable Addition (BVA) [MHB13].

Consider a set of literals M_{lit} and a set of clauses $M_{cls} \subseteq F$, such that for all $\ell \in M_{lit}$ and $C \in M_{cls}$, each clause $(C \setminus M_{lit} \cup \{\ell\})$ is either in F or a tautology. Then an application of BVA adds the clauses $S_x = \{(\ell \vee x) \mid l \in M_{lit}\}$ and $S_{\overline{x}} = \{(C \setminus M_{lit}) \cup \{\overline{x}\} \mid C \in M_{cls}\}$, and removes the clauses $C \setminus M_{lit}$.

An application of BVA is logged as follows: (1) Add the constraint PB(C) for each $C \in S_{\overline{x}}$ with the witness $\{x \rightarrow 0\}$. (2) Add the constraint PB(C) for each

$C \in S_x$ with the witness $\{x \to 1\}$. (3) Delete each constraint PB(C) for $C \in M_{cls}$ as a RUP constraint.

Structure-based Labelling [KBSJ17].

Given an objective variable b and a clause C that is blocked on the literal ℓ, when $b = 1$, an application of structure-based labelling replaces C with $C \vee b$. The proof is logged as follows: (1) Introduce the constraint PB($C \vee b$) that is RUP. (2) Delete the constraint PB(C) with the witness $\{\ell \to 1\}$.

Failed Literal Elimination (FLE) [Fre95, LB01, ZM88].

A literal ℓ is failed (denoted $\ell \vdash_{\text{UP}} \bot$) if setting $\ell = 1$ allows unit propagation to derive a conflict (i.e., an empty clause). An application of FLE fixes $\ell = 0$ when ℓ is a failed literal for which $\overline{\ell}$ is not in the objective.

In addition to standard FLE, MaxPre implements an extension that also fixes a literal $\ell = 0$ if: (i) $\overline{\ell}$ is not in the objective function (ii) each clause in F that contains ℓ also contains some other literal ℓ' that is implied by ℓ by unit propagation (denoted $\ell \vdash_{\text{UP}} \ell'$), i.e., setting $\ell = 1$ also fixes $\ell' = 1$ after a sequence of unit propagation steps is applied.

Logging FLE. For a failed literal ℓ the constraint $\overline{\ell} \geq 1$ is RUP. For the extended technique the constraint $\overline{\ell} \geq 1$ is introduced with the witness $\{\ell \to 0\}$. Afterwards the generic procedure for fixing literals described in Section A.1 is invoked.

Implied Literal Detection.

If both a literal ℓ_1 and its negation $\overline{\ell}_1$ imply another literal ℓ_2 by unit propagation (i.e., propagating either $\ell = 1$ or $\ell = 0$ also propagates $\ell_2 = 1$), the preprocessor fixes $\ell_2 = 1$.

As an extension to this technique, the preprocessor also fixes $\ell_2 = 1$ if (i) ℓ_1 implies ℓ_2 by unit propagation, (ii) neither ℓ_1 nor ℓ_2 appear in the objective function in either polarity, and (iii) each clause containing $\overline{\ell}_2$ also contains some other literal ℓ' that is implied by $\overline{\ell}_1$ by unit propagation.

Logging Implied Literals. For some intuition, note that $\ell_1 \vdash_{\text{UP}} \ell_2$ does not in general imply $\overline{\ell}_2 \vdash_{\text{UP}} \overline{\ell}_1$. Thus, there is no guarantee that $\ell_2 \geq 1$ would be RUP. Given that $\ell_1 \vdash_{\text{UP}} \ell_2$ and $\overline{\ell}_1 \vdash_{\text{UP}} \ell_2$, the proof is instead logged as follows:

(1) Add $\overline{\ell}_1 + \ell_2 \geq 1$ and $\ell_1 + \ell_2 \geq 1$ that are both RUP.

(2) Introduce the constraint $\ell_2 \geq 1$ by divide the sum of constraints introduced in step (1) by 2. Move the new constraint to the core constraints.

(3) Delete the constraints introduced in step (1).

(4) Invoke the generic procedure detailed in Section A.1 to fix $\ell_2 = 1$.

The extended technique is logged by first adding the constraint $\ell_1 + \ell_2 \geq 1$ with the witness $\{\ell_2 \to 1\}$. For some intuition, if the constraint is falsified, the assumptions guarantee that $\ell' = 1$ so the value of ℓ_2 can be flipped without falsifying other constraints.

Equivalent Literal Substitution [Bra04, Li00, VG05].

If $\ell_1 \vdash_{\text{UP}} \ell_2$ and $\overline{\ell}_1 \vdash_{\text{UP}} \overline{\ell}_2$, the equivalent literal technique substitutes ℓ_1 with ℓ_2. As an extension to this technique, the same substitution is applied also in cases where the following three conditions hold: (i) $\ell_1 \vdash_{\text{UP}} \ell_2$, (ii) neither ℓ_1 nor ℓ_2 appear in the objective function in either polarity, and (iii) $\overline{\ell}_1$ implies some other literal in each clause containing $\overline{\ell}_2$ by unit propagation.

Logging Equivalent Literals. An application of equivalent literal substitution is logged as follows.

(1) Introduce the clauses $\overline{\ell}_1 + \ell_2 \geq 1$ and $\ell_1 + \overline{\ell}_2 \geq 1$ as RUP. In the case of the extended technique, $\ell_1 + \overline{\ell}_2 \geq 1$ is added with the witness $\{\ell_2 \to 0\}$.

(2) For each clause $C \vee \ell_1$, replace $\mathrm{PB}(C \vee \ell_1)$ with $\mathrm{PB}(C \vee \ell_2)$ with the RUP rule.

(3) For each clause $C \vee \overline{\ell}_1$, replace $\mathrm{PB}(C \vee \overline{\ell}_1)$ with $\mathrm{PB}(C \vee \overline{\ell}_2)$ with the RUP rule.

(4) If ℓ_1 or $\overline{\ell}_1$ appear in the objective function, replace them with ℓ_2 and $\overline{\ell}_2$, respectively.

(5) Remove the constraints introduced in step (1).

Hardening [ABGL12, IBJ22, MHM12].

Given an upper bound UB for the optimal cost of (F, O) and an objective variable b that has a coefficient $w^b > UB$ in O, hardening fixes $b = 0$. Proof logging for hardening has been previously studied in [BBN+23]. In [BBN+23], however, the hardening is done with the presence of so-called objective-improving constraints, i.e., constraints of form $O \leq UB - 1$, where UB is the cost of the best currently known solution. In the context of preprocessing where the preprocessor should provide an equioptimal instance as an output, introducing objective-improving constraints to the instance is not possible. Instead, given a solution ρ to F with cost $O(\rho) = UB$ and an objective variable b with $w^b > UB$, we introduce the constraint $\overline{b} \geq 1$ with ρ as the witness and then invoke the generic procedure for fixing variables, as detailed in Section A.1.

A.4 Conversion to WCNF — Renaming Variables

In the final stage of preprocessing, MaxPre converts the instance to WCNF. The conversion removes the objective constant as described in Section 3.1 of the main paper. Additionally, the conversion 'renames' (some of) the variables.

There are two reasons for renaming variables. The first is to remove any gaps in the indexing of variables. In WCNF, variables are named with integers. During preprocessing, some variables in the instance might have been eliminated from the instance. At the end MaxPre compacts the range of variables to be continuous and start from 1. The second reason for renaming variables is to sync names between WCNF and the pseudo-Boolean proof. In the pseudo-Boolean proofs, the naming scheme of variables is different, valid variable names include, for instance, `x1`, `x2`, `y15`, `_b4`. When a WCNF instance is converted to a pseudo-Boolean instance, the variable `i` of the WCNF instance is mapped to the variable `xi` of the pseudo-Boolean instance. For jth non-unit soft clause of a WCNF instance, the conversion introduces a variable `_bj`. During preprocessing, the 'proof logger' of MaxPre takes care of mapping MaxPre variables to correct variable names in proof. In the end, however, MaxPre produces an output WCNF file, and at this point, each variable `i` of WCNF instance should again correspond to variable `xi` of proof. Thus, for example, all `_b`-variables are replaced with `x`-variables.

Logging variable naming. Assume that the instance has a set of variables V and for each $x \in V$, we wish to use name $f(x)$ instead of x in the end. We do proof logging for variable renaming in two phases. (1) For each $x \in V$, introduce temporary variable t_x, set $x = t_x$ and then 'move' all the constraints and the objective function to the temporary namespace. The original constraints and encodings for $x = t_x$ are then removed. (2) For each $x \in V$, introduce $f(x) = t_x$, and 'move' the constraints and the objective to the final namespace. The temporary constraints and encodings are then removed.

A.5 On Solution Reconstruction and Instances Solved During Preprocessing

Finally, we note that while the focus of this work has been on certifying the preservation of the costs of solutions, in practice our certified preprocessor also allows reconstructing a minimum-cost solution to the input. More precisely, consider an input WCNF instance $\mathcal{F}^W$, a preprocessed instance $\mathcal{F}_P^W$, and an optimal solution ρ_p to $\mathcal{F}_P^W$. Then MaxPre can compute an optimal solution ρ to $\mathcal{F}^W$ in linear time with respect to the number of preprocessing steps performed. More details can be found in [KBSJ17].

Importantly, the optimality of a reconstructed solution can be easily verified without considering how the reconstruction is implemented in practice; given that we have verified the equioptimality of $\mathcal{F}^W$ and $\mathcal{F}_P^W$, and that ρ_p is an optimal solution to $\mathcal{F}_P^W$, the optimality of reconstructed ρ to $\mathcal{F}^W$ can be verified by checking that (i) ρ indeed is a solution to $\mathcal{F}^W$ (ii) The cost of ρ w.r.t. $\mathcal{F}^W$ is equivalent to the cost of ρ_p w.r.t. $\mathcal{F}_P^W$.

On a related note, MaxPre can actually solve some instances during preprocessing, either by: (i) determining that the hard clauses do not have solutions, or (ii) computing an optimal solution to some working instance. In practice (i) happens by the derivation of the unsatisfiable empty (hard) clause and (ii) by the removal of every single clause from the working instance. We have designed the preprocessor

to always terminate with an output WCNF and a proof of equioptimality rather than producing different kinds of proofs.

If an empty hard clause is derived, the preprocessing is immediately terminated and an output WCNF instance containing a single hard empty clause produced. Additionally, an empty constraint $0 \geq 1$ is added to the proof and all other core constraints deleted by the RUP rule. Notice how the proof of equioptimality between the input and output can in this case be seen as a proof of infeasibility of the input hard clauses.

If all clauses are removed from the working instance, MaxPRE terminates and outputs the instance obtained after constant removal (recall Stage 5 in Section 3) on an instance without other clauses.

References

[ABGL12] Carlos Ansótegui, María Luisa Bonet, Joel Gabàs, and Jordi Levy. Improving SAT-based weighted MaxSAT solvers. In *Proceedings of the 18th International Conference on Principles and Practice of Constraint Programming (CP '12)*, volume 7514 of *Lecture Notes in Computer Science*, pages 86–101. Springer, October 2012.

[ABM+11] Eyad Alkassar, Sascha Böhme, Kurt Mehlhorn, Christine Rizkallah, and Pascal Schweitzer. An introduction to certifying algorithms. *it - Information Technology Methoden und innovative Anwendungen der Informatik und Informationstechnik*, 53(6):287–293, December 2011.

[BBN+23] Jeremias Berg, Bart Bogaerts, Jakob Nordström, Andy Oertel, and Dieter Vandesande. Certified core-guided MaxSAT solving. In *Proceedings of the 29th International Conference on Automated Deduction (CADE-29)*, volume 14132 of *Lecture Notes in Computer Science*, pages 1–22. Springer, July 2023.

[BBP20] Jeremias Berg, Fahiem Bacchus, and Alex Poole. Abstract cores in implicit hitting set MaxSat solving. In *Proceedings of the 23rd International Conference on Theory and Applications of Satisfiability Testing (SAT '20)*, volume 12178 of *Lecture Notes in Computer Science*, pages 277–294. Springer, July 2020.

[BGMN23] Bart Bogaerts, Stephan Gocht, Ciaran McCreesh, and Jakob Nordström. Certified dominance and symmetry breaking for combinatorial optimisation. *Journal of Artificial Intelligence Research*, 77:1539–1589, August 2023. Preliminary version in *AAAI '22*.

[BHvMW21] Armin Biere, Marijn J. H. Heule, Hans van Maaren, and Toby Walsh, editors. *Handbook of Satisfiability*, volume 336 of *Frontiers in Artificial Intelligence and Applications*. IOS Press, 2nd edition, February 2021.

[Bie06] Armin Biere. Tracecheck. `http://fmv.jku.at/tracecheck/`, 2006.

[BJ19] Jeremias Berg and Matti Järvisalo. Unifying reasoning and core-guided search for maximum satisfiability. In *Proceedings of the 16th European Conference on Logics in Artificial Intelligence (JELIA '19)*, volume 11468 of *Lecture Notes in Computer Science*, pages 287–303. Springer, 2019.

[BLB10] Robert Brummayer, Florian Lonsing, and Armin Biere. Automated testing and debugging of SAT and QBF solvers. In *Proceedings of the 13th International Conference on Theory and Applications of Satisfiability Testing (SAT '10)*, volume 6175 of *Lecture Notes in Computer Science*, pages 44–57. Springer, July 2010.

[BLM07] Maria Luisa Bonet, Jordi Levy, and Felip Manyà. Resolution for Max-SAT. *Artificial Intelligence*, 171(8-9):606–618, 2007.

[BMM+23] Bart Bogaerts, Ciaran McCreesh, Magnus O. Myreen, Jakob Nordström, Andy Oertel, and Yong Kiam Tan. Documentation of VeriPB and CakePB for the SAT competition 2023. Available at `https://satcompetition.github.io/2023/checkers.html`, March 2023.

[BN21] Samuel R. Buss and Jakob Nordström. Proof complexity and SAT solving. In Biere et al. [BHvMW21], chapter 7, pages 233–350.

[Bra04] Ronen I. Brafman. A simplifier for propositional formulas with many binary clauses. *IEEE Transactions on Systems, Man, and Cybernetics, Part B (Cybernetics)*, 34(1):52–59, 2004.

[BSJ16] Jeremias Berg, Paul Saikko, and Matti Järvisalo. Subsumed label elimination for maximum satisfiability. In *Proceedings of the 22nd European Conference on Artificial Intelligence (ECAI '16)*, volume 285 of *FAIA*, pages 630–638. IOS Press, 2016.

[CCT87] William Cook, Collette Rene Coullard, and György Turán. On the complexity of cutting-plane proofs. *Discrete Applied Mathematics*, 18(1):25–38, November 1987.

[CHH+17] Luís Cruz-Filipe, Marijn J. H. Heule, Warren A. Hunt Jr., Matt Kaufmann, and Peter Schneider-Kamp. Efficient certified RAT verification. In *Proceedings of the 26th International Conference on Automated Deduction (CADE-26)*, volume 10395 of *Lecture Notes in Computer Science*, pages 220–236. Springer, August 2017.

[CMS17] Luís Cruz-Filipe, João P. Marques-Silva, and Peter Schneider-Kamp. Efficient certified resolution proof checking. In *Proceedings of the 23rd International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS '17)*, volume 10205 of *Lecture Notes in Computer Science*, pages 118–135. Springer, April 2017.

[DB11] Jessica Davies and Fahiem Bacchus. Solving MAXSAT by solving a sequence of simpler SAT instances. In *Proceedings of the 17th International Conference on Principles and Practice of Constraint Programming*

(CP '11), volume 6876 of *Lecture Notes in Computer Science*, pages 225–239. Springer, September 2011.

[DB13] Jessica Davies and Fahiem Bacchus. Exploiting the power of MIP solvers in MAXSAT. In *Proceedings of the 16th International Conference on Theory and Applications of Satisfiability Testing (SAT '13)*, volume 7962 of *Lecture Notes in Computer Science*, pages 166–181. Springer, July 2013.

[EB05] Niklas Eén and Armin Biere. Effective preprocessing in SAT through variable and clause elimination. In *Proceedings of the 8th International Conference on Theory and Applications of Satisfiability Testing (SAT '05)*, volume 3569 of *Lecture Notes in Computer Science*, pages 61–75. Springer, June 2005.

[EGMN20] Jan Elffers, Stephan Gocht, Ciaran McCreesh, and Jakob Nordström. Justifying all differences using pseudo-Boolean reasoning. In *Proceedings of the 34th AAAI Conference on Artificial Intelligence (AAAI '20)*, pages 1486–1494, February 2020.

[ES03] Niklas Eén and Niklas Sörensson. Temporal induction by incremental SAT solving. In Ofer Strichman and Armin Biere, editors, *First International Workshop on Bounded Model Checking, (BMC '03)*, volume 89 of *Electronic Notes in Theoretical Computer Science*, pages 543–560. Elsevier, 2003.

[ES06] Niklas Eén and Niklas Sörensson. Translating pseudo-Boolean constraints into SAT. *Journal on Satisfiability, Boolean Modeling and Computation*, 2(1-4):1–26, March 2006.

[FMSV20] Yuval Filmus, Meena Mahajan, Gaurav Sood, and Marc Vinyals. MaxSAT resolution and subcube sums. In *Proceedings of the 23rd International Conference on Theory and Applications of Satisfiability Testing (SAT '20)*, volume 12178 of *Lecture Notes in Computer Science*, pages 295–311. Springer, July 2020.

[Fre95] Jon William Freeman. *Improvements to Propositional Satisfiability Search Algorithms*. PhD thesis, University of Pennsylvania, 1995.

[Gim64] James F. Gimpel. A reduction technique for prime implicant tables. In *Proceedings of the 5th Annual Symposium on Switching Circuit Theory and Logical Design, (SWCT '64)*, pages 183–191. IEEE Computer Society, 1964.

[GMKN17] Armaël Guéneau, Magnus O. Myreen, Ramana Kumar, and Michael Norrish. Verified characteristic formulae for CakeML. In *Proceedings of the 26th European Symposium on Programming (ESOP '17)*, volume 10201 of *Lecture Notes in Computer Science*, pages 584–610. Springer, April 2017.

[GMM+20] Stephan Gocht, Ross McBride, Ciaran McCreesh, Jakob Nordström, Patrick Prosser, and James Trimble. Certifying solvers for clique and maximum common (connected) subgraph problems. In *Proceedings of the 26th International Conference on Principles and Practice of Constraint Programming (CP '20)*, volume 12333 of *Lecture Notes in Computer Science*, pages 338–357. Springer, September 2020.

[GMM+24] Stephan Gocht, Ciaran McCreesh, Magnus O. Myreen, Jakob Nordström, Andy Oertel, and Yong Kiam Tan. End-to-end verification for subgraph solving. In *Proceedings of the 368h AAAI Conference on Artificial Intelligence (AAAI '24)*, pages 8038–8047, February 2024.

[GMN20] Stephan Gocht, Ciaran McCreesh, and Jakob Nordström. Subgraph isomorphism meets cutting planes: Solving with certified solutions. In *Proceedings of the 29th International Joint Conference on Artificial Intelligence (IJCAI '20)*, pages 1134–1140, July 2020.

[GMN22] Stephan Gocht, Ciaran McCreesh, and Jakob Nordström. An auditable constraint programming solver. In *Proceedings of the 28th International Conference on Principles and Practice of Constraint Programming (CP '22)*, volume 235 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 25:1–25:18, August 2022.

[GMNO22] Stephan Gocht, Ruben Martins, Jakob Nordström, and Andy Oertel. Certified CNF translations for pseudo-Boolean solving. In *Proceedings of the 25th International Conference on Theory and Applications of Satisfiability Testing (SAT '22)*, volume 236 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 16:1–16:25, August 2022.

[GN03] Evgueni Goldberg and Yakov Novikov. Verification of proofs of unsatisfiability for CNF formulas. In *Proceedings of the Conference on Design, Automation and Test in Europe (DATE '03)*, pages 886–891, March 2003.

[GN21] Stephan Gocht and Jakob Nordström. Certifying parity reasoning efficiently using pseudo-Boolean proofs. In *Proceedings of the 35th AAAI Conference on Artificial Intelligence (AAAI '21)*, pages 3768–3777, February 2021.

[Goc22] Stephan Gocht. *Certifying Correctness for Combinatorial Algorithms by Using Pseudo-Boolean Reasoning*. PhD thesis, Lund University, June 2022. Available at `https://portal.research.lu.se/en/publications/certifying-correctness-for-combinatorial-algorithms-by-using-pseu`.

[HHW13a] Marijn J. H. Heule, Warren A. Hunt Jr., and Nathan Wetzler. Trimming while checking clausal proofs. In *Proceedings of the 13th International Conference on Formal Methods in Computer-Aided Design (FMCAD '13)*, pages 181–188, October 2013.

[HHW13b] Marijn J. H. Heule, Warren A. Hunt Jr., and Nathan Wetzler. Verifying refutations with extended resolution. In *Proceedings of the 24th International Conference on Automated Deduction (CADE-24)*, volume 7898 of *Lecture Notes in Computer Science*, pages 345–359. Springer, June 2013.

[HOGN24] Alexander Hoen, Andy Oertel, Ambros Gleixner, and Jakob Nordström. Certifying MIP-based presolve reductions for 0–1 integer linear programs. In *Proceedings of the 21st International Conference on the Integration of Constraint Programming, Artificial Intelligence, and Operations Research (CPAIOR '24)*, May 2024. To appear.

[IBJ22] Hannes Ihalainen, Jeremias Berg, and Matti Järvisalo. Clause redundancy and preprocessing in maximum satisfiability. In *Proceedings of the 11th International Joint Conference on Automated Reasoning (IJCAR '22)*, volume 13385 of *Lecture Notes in Computer Science*, pages 75–94. Springer, August 2022.

[IMM19] Alexey Ignatiev, António Morgado, and João Marques-Silva. RC2: an efficient MaxSAT solver. *Journal on Satisfiability, Boolean Modeling and Computation*, 11(1):53–64, 2019.

[IOT+24] Hannes Ihalainen, Andy Oertel, Yong Kiam Tan, Jeremias Berg, Matti Järvisalo, Magnus O. Myreen, and Jakob Nordström. Experimental Repository for "Certified MaxSAT Preprocessing", February 2024.

[JBH10] Matti Järvisalo, Armin Biere, and Marijn Heule. Blocked clause elimination. In *Proceedings of the 16th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS '10)*, volume 6015 of *Lecture Notes in Computer Science*, pages 129–144. Springer, 2010.

[JBIJ23] Christoph Jabs, Jeremias Berg, Hannes Ihalainen, and Matti Järvisalo. Preprocessing in SAT-based multi-objective combinatorial optimization. In *Proceedings of the 29th International Conference on Principles and Practice of Constraint Programming (CP '23)*, volume 280 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 18:1–18:20, 2023.

[KBSJ17] Tuukka Korhonen, Jeremias Berg, Paul Saikko, and Matti Järvisalo. MaxPre: An extended MaxSAT preprocessor. In *Proceedings of the 20th International Conference on Theory and Applications of Satisfiability Testing (SAT '17)*, volume 10491 of *Lecture Notes in Computer Science*, pages 449–456. Springer, 2017.

[LB01] Daniel Le Berre. Exploiting the real power of unit propagation lookahead. *Electronic Notes in Discrete Mathematics*, 9:59–80, 2001.

[Li00] Chu Min Li. Integrating equivalency reasoning into Davis-Putnam procedure. In *Proceedings of the 17th National Conference on Artificial Intelligence and 12th Conference on Innovative Applications of Artificial Intelligence*, pages 291–296. AAAI Press / The MIT Press, 2000.

[LNOR11] Javier Larrosa, Robert Nieuwenhuis, Albert Oliveras, and Enric Rodríguez-Carbonell. A framework for certified Boolean branch-and-bound optimization. *Journal of Automated Reasoning*, 46(1):81–102, 2011.

[LP10] Daniel Le Berre and Anne Parrain. The Sat4j library, release 2.2. *Journal on Satisfiability, Boolean Modeling and Computation*, 7:59–64, July 2010.

[Maxa] MaxPre 2 : MaxSAT preprocessor. `https://bitbucket.org/coreo-group/maxpre2`.

[Maxb] MaxSAT evaluations: Evaluating the state of the art in maximum satisfiability solver technology. `https://maxsat-evaluations.github.io/`.

[Max23] MaxSAT evaluation 2023. `https://maxsat-evaluations.github.io/2023`, July 2023.

[MHB13] Norbert Manthey, Marijn J. H. Heule, and Armin Biere. Automated reencoding of Boolean formulas. In *8th International Haifa Verification Conference (HVC '12), Revised Selected Papers*, volume 7857 of *Lecture Notes in Computer Science*, pages 102–117. Springer, 2013.

[MHM12] António Morgado, Federico Heras, and João Marques-Silva. Improvements to core-guided binary search for MaxSAT. In *Proceedings of the 15th International Conference on Theory and Applications of Satisfiability Testing (SAT '12)*, volume 7317 of *Lecture Notes in Computer Science*, pages 284–297. Springer, 2012.

[MIB+19] António Morgado, Alexey Ignatiev, María Luisa Bonet, João P. Marques-Silva, and Samuel R. Buss. DRMaxSAT with MaxHS: First contact. In *Proceedings of the 22nd International Conference on Theory and Applications of Satisfiability Testing (SAT '19)*, volume 11628 of *Lecture Notes in Computer Science*, pages 239–249. Springer, July 2019.

[MLM21] João Marques-Silva, Inês Lynce, and Sharad Malik. Conflict-driven clause learning SAT solvers. In Armin Biere, Marijn Heule, Hans van Maaren, and Toby Walsh, editors, *Handbook of Satisfiability - Second Edition*, volume 336 of *Frontiers in Artificial Intelligence and Applications*, pages 133–182. IOS Press, 2021.

[MM11] António Morgado and João Marques-Silva. On validating Boolean optimizers. In *Proceedings of the 23rd IEEE International Conference on Tools with Artificial Intelligence, (ICTAI '11)*, pages 924–926, 2011.

[MM23] Matthew McIlree and Ciaran McCreesh. Proof logging for smart extensional constraints. In *Proceedings of the 29th International Conference on Principles and Practice of Constraint Programming (CP '23)*, volume 280 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 26:1–26:17, August 2023.

[MMN24] Matthew McIlree, Ciaran McCreesh, and Jakob Nordström. Proof logging for the circuit constraint. In *Proceedings of the 21st International Conference on the Integration of Constraint Programming, Artificial Intelligence, and Operations Research (CPAIOR '24)*, May 2024. To appear.

[MMNS11] Ross M. McConnell, Kurt Mehlhorn, Stefan Näher, and Pascal Schweitzer. Certifying algorithms. *Computer Science Review*, 5(2):119–161, May 2011.

[MO14] Magnus O. Myreen and Scott Owens. Proof-producing translation of higher-order logic into pure and stateful ML. *Journal of Functional Programming*, 24(2–3):284–315, January 2014.

[OGMS02] Richard Ostrowski, Éric Grégoire, Bertrand Mazure, and Lakhdar Sais. Recovering and exploiting structural knowledge from CNF formulas. In *Proceedings of the 8th International Conference on Principles and Practice of Constraint Programming (CP '02)*, volume 2470 of *Lecture Notes in Computer Science*, pages 185–199. Springer, 2002.

[PCH20] Matthieu Py, Mohamed Sami Cherif, and Djamal Habet. Towards bridging the gap between SAT and Max-SAT refutations. In *Proceedings of the 32nd IEEE International Conference on Tools with Artificial Intelligence (ICTAI '20)*, pages 137–144, November 2020.

[PCH21] Matthieu Py, Mohamed Sami Cherif, and Djamal Habet. A proof builder for Max-SAT. In *Proceedings of the 24th International Conference on Theory and Applications of Satisfiability Testing (SAT '21)*, volume 12831 of *Lecture Notes in Computer Science*, pages 488–498. Springer, July 2021.

[PCH22] Matthieu Py, Mohamed Sami Cherif, and Djamal Habet. Proofs and certificates for Max-SAT. *Journal of Artificial Intelligence Research*, 75:1373–1400, December 2022.

[PRB18] Tobias Paxian, Sven Reimer, and Bernd Becker. Dynamic polynomial watchdog encoding for solving weighted MaxSAT. In *Proceedings of the 21st International Conference on Theory and Applications of Satisfiability Testing (SAT '18)*, volume 10929 of *Lecture Notes in Computer Science*, pages 37–53. Springer, July 2018.

[PRB21] Tobias Paxian, Pascal Raiola, and Bernd Becker. On preprocessing for weighted MaxSAT. In *Proceedings of the 22nd International Conference on Verification, Model Checking, and Abstract Interpretation, (VMCAI '21)*, volume 12597 of *Lecture Notes in Computer Science*, pages 556–577. Springer, 2021.

[SAT] The International SAT Competitions web page. `http://www.satcompetition.org`.

[SN08] Konrad Slind and Michael Norrish. A brief overview of HOL4. In *Proceedings of the 21st International Conference on Theorem Proving in Higher Order Logics (TPHOLs '08)*, volume 5170 of *Lecture Notes in Computer Science*, pages 28–32. Springer, August 2008.

[SP04] Sathiamoorthy Subbarayan and Dhiraj K. Pradhan. NiVER: Non-increasing variable elimination resolution for preprocessing SAT instances. In *Proceedings of the 7th International Conference on Theory and Applications of Satisfiability Testing (SAT '04)*, volume 3542 of *Lecture Notes in Computer Science*, pages 276–291. Springer, 2004.

[THM23] Yong Kiam Tan, Marijn J. H. Heule, and Magnus O. Myreen. Verified propagation redundancy and compositional UNSAT checking in CakeML. *International Journal on Software Tools for Technology Transfer*, 25:167–184, February 2023. Preliminary version in *TACAS '21*.

[TMK+19] Yong Kiam Tan, Magnus O. Myreen, Ramana Kumar, Anthony C. J. Fox, Scott Owens, and Michael Norrish. The verified CakeML compiler backend. *Journal of Functional Programming*, 29:e2:1–e2:57, February 2019.

[VDB22] Dieter Vandesande, Wolf De Wulf, and Bart Bogaerts. QMaxSATpb: A certified MaxSAT solver. In *Proceedings of the 16th International Conference on Logic Programming and Non-monotonic Reasoning (LP-NMR '22)*, volume 13416 of *Lecture Notes in Computer Science*, pages 429–442. Springer, September 2022.

[Ver] VeriPB: Verifier for pseudo-Boolean proofs. `https://gitlab.com/MIAOresearch/software/VeriPB`.

[VG05] Allen Van Gelder. Toward leaner binary-clause reasoning in a satisfiability solver. *Annals of Mathematics and Artificial Intelligence*, 43(1):239–253, 2005.

[WHH14] Nathan Wetzler, Marijn J. H. Heule, and Warren A. Hunt Jr. DRAT-trim: Efficient checking and trimming using expressive clausal proofs. In *Proceedings of the 17th International Conference on Theory and Applications of Satisfiability Testing (SAT '14)*, volume 8561 of *Lecture Notes in Computer Science*, pages 422–429. Springer, July 2014.

[ZM88] Ramin Zabih and David A. McAllester. A rearrangement search strategy for determining propositional satisfiability. In *Proceedings of the 7th National Conference on Artificial Intelligence (AAAI '88)*, pages 155–160. AAAI Press / The MIT Press, 1988.

Faster Certified Symmetry Breaking Using Orders With Auxiliary Variables

Abstract

Symmetry breaking is a crucial technique in modern combinatorial solving, but it is difficult to be sure it is implemented correctly. The most successful approach to deal with bugs is to make solvers *certifying*, so that they output not just a solution, but also a mathematical proof of correctness in a standard format, which can then be checked by a formally verified checker. This requires justifying symmetry reasoning within the proof, but developing efficient methods for this has remained a long-standing open challenge. A fully general approach was recently proposed by Bogaerts et al. (2023), but it relies on encoding lexicographic orders with big integers, which quickly becomes infeasible for large symmetries. In this work, we develop a method for instead encoding orders with auxiliary variables. We show that this leads to orders-of-magnitude speed-ups in both theory and practice by running experiments on proof logging and checking for SAT symmetry breaking using the state-of-the-art SATSUMAsymmetry breaker and the VERIPB proof checking toolchain.

1 Introduction

An important challenge in combinatorial solving is to avoid repeatedly exploring different parts of the search space that are equivalent under symmetries. In a wide range of combinatorial solving paradigms, *symmetry breaking* is deployed as a default technique, including mixed integer programming [AW13, BBB+24] and constraint programming [Wal06]. The importance of symmetry breaking is supported by both theoretical considerations [Urq99] and experimental results [PR19]. A detailed discussion of symmetry breaking is given, e.g., by [Sak21].

Markus Anders, Bart Bogaerts, Benjamin Bogø, Arthur Gontier, Wietze Koops, Ciaran McCreesh, Magnus O. Myreen, Jakob Nordström, Andy Oertel, Adrian Rebola-Pardo, and Yong Kiam Tan. "Faster Certified Symmetry Breaking Using Orders With Auxiliary Variables". In *Proceedings of the 40th Annual AAAI Conference on Artificial Intelligence (AAAI '26)*, January 2026.

Symmetry breaking has not been adopted as mainstream in Boolean satisfiability (SAT) solving, however, despite a body of work [AMS03, DBBD16, ABR24] showing the potential for speed-ups also in this setting. One reason for this could perhaps be the higher cost, relatively speaking, of symmetry breaking compared to low-level SAT reasoning, but state-of-the art symmetry detection is efficient enough to use by default without degrading performance [ABR24]. A more important concern is that the SAT community places a strong emphasis on provable correctness. For over a decade, SAT solvers taking part in the annual SAT competitions have had to generate machine-verifiable proofs for their results. Such proofs are especially important for sophisticated techniques such as symmetry breaking, which is notoriously difficult to implement correctly. However, except for some special cases [HHW15], it has not been known how to generate proofs for symmetry breaking in the DRAT proof format [WHH14] used in the competitions, or whether this is even possible.

The way symmetry breaking is typically done in SAT solving is by introducing *lex-leader* constraints, which are encoded as the clauses

$$\begin{array}{lll} s_1 \vee \overline{x}_1 & s_1 \vee y_1 & y_1 \vee \overline{x}_1 \\ s_{i+1} \vee \overline{s}_i \vee \overline{x}_{i+1} & s_{i+1} \vee \overline{s}_i \vee y_{i+1} & \overline{s}_i \vee y_{i+1} \vee \overline{x}_{i+1} \end{array} \tag{1}$$

that can be thought of as encoding a circuit enforcing $(x_1, \ldots, x_n) \preceq_{\text{lex}} (y_1, \ldots, y_n)$—here, s_i are fresh auxiliary variables encoding that the x- and y-variables are equal up to position i; using these, we enforce that x_i is false and y_i true the first time this does not hold. Such clauses are clearly not implied by the original formula, and the problem is how to prove that they can be added without changing the satisfiability of the input. Although the RAT rule [JHB12] in DRAT can handle a single symmetry [KT24], once the first symmetry is broken it is not known how or even if the other symmetries found by the symmetry breaker could be proven correct using DRAT.

[BGMN23] finally resolved this long-standing open problem by introducing a stronger proof format, which operates with *pseudo-Boolean* (i.e., 0–1 linear) inequalities rather than clauses, and reasons in terms of *dominance* [CS15] to support fully general symmetry breaking without any limitations on the number of symmetries that can be handled. One benefit of this richer format is that a single inequality

$$2^{n-1}x_1 + \cdots + 2x_{n-1} + x_n \leq 2^{n-1}y_1 + \cdots + 2y_{n-1} + y_n \, , \tag{2}$$

can be used in the proof to encode lexicographic order, and from this constraint it is straightforward to derive the clauses (1) used by the solver. However, at least n^2 bits are needed to represent the coefficients in (2), while the representation of (1) scales linearly with n. This means that proof generation incurs a linear overhead compared to solving. Also, the algorithm by [ABR24] can break a symmetry in quasi-linear time measured in the number of variables k remapped by the symmetry, which introduces yet another asymptotic slowdown in proof generation if $k \ll n$. Furthermore, the exponentially growing integer coefficients in (2) require expensive arbitrary-precision arithmetic, which slows down proof checking. All of these problems combine to make the proof logging approach

proposed by [BGMN23] infeasible for large-scale problems requiring non-trivial symmetry breaking.

In this work, we present an asymptotically faster method for generating and checking proofs of correctness for symmetry breaking. The main new technical idea is to use *auxiliary variables* to encode the lexicographic order used for the dominance reasoning, similar to the clausal encoding in (1). Unfortunately, this breaks the fundamental invariant of [BGMN23] that all low-level proofs should be implicational. When one needs to prove that a symmetry-breaking constraint respects lexicographical order, the encoding of this order will contain auxiliary variables that are not mentioned in the premises, and so this property cannot possibly be implied. We therefore need to make a substantial redesign of the proof system of [BGMN23] to work with auxiliary variables. Very briefly, our key technical twist is to split the encoding of the order into two parts, putting one part into the premises, so that the property of implicational low-level proofs can be maintained. Our redesigned proof system supports fully general symmetry breaking in a similar fashion to [BGMN23], but is significantly more efficient. Specifically, we prove that our approach leads to asymptotic gains for proof logging and checking for symmetry breaking by at least a linear factor in the size n of the lexicographic order used.

We have implemented support for our new proof system in the proof checker VeriPB [BGMN23, GN21, Goc22] and in its formally verified checking backend CakePB [GMM+24]. Together, these yield an efficient, end-to-end verified proof checking toolchain for symmetry breaking proofs. We have also enhanced the state-of-the-art SAT symmetry breaker satsuma [ABR24] to generate proofs of correctness in our new format as well as that of [BGMN23] for a comparative evaluation of performance. Our experimental findings match our theoretical results and show that only a constant overhead in running time is required for proof logging with our new method. Proof checking performance is also vastly better compared to [BGMN23], although here there might be room for further improvements.

Our paper is organized as follows. After reviewing preliminaries in Section 2, we present our new proof logging system in Section 3. Sections 4 and 5 discuss how proof logging and checking can be improved asymptotically using our new method, which is confirmed by our experiments in Section 6. We conclude with a brief discussion of future work in Section 7. In this full-length version, we also provide five appendices. Appendix A gives an overview of the cutting planes proof system and the syntax used by VeriPB for implicational reasoning. Appendix B provides further details on the extended proof system introduced in Sections 3.1 up to 3.5, including all proofs. Appendix C provides all details on how the proof logging is implemented in satsuma. Appendix D presents a concrete toy example of a VeriPB proof generated by satsuma. Appendix E contains an overview of the crafted benchmarks that we used in our experimental evaluation.

2 Preliminaries

We start with a brief review of pseudo-Boolean reasoning. For more details, we refer the reader to, e.g., [BN21] or [BGMN23]. A *Boolean variable* takes values 0 or 1. A *literal* over a Boolean variable x is x itself or its negation $\overline{x} = 1 - x$. A *pseudo-Boolean (PB) constraint* C is an integer linear inequality over literals

$$C \doteq \sum_i a_i \ell_i \geq A\,, \tag{3}$$

where we use $\doteq$ to denote syntactic equivalence. Without loss of generality the coefficients a_i and the right-hand side A are non-negative and the literals ℓ_i are over distinct variables. The *trivially false constraint* is $\bot \doteq 0 \geq 1$. The *negation* $\neg C$ of the pseudo-Boolean constraint C in (3) is the pseudo-Boolean constraint $\neg C \doteq \sum_i a_i \overline{\ell}_i \geq \sum_i a_i - A + 1$. A *pseudo-Boolean formula* F is a conjunction $F \doteq \bigwedge_i C_i$ or equivalently a set $F \doteq \bigcup_i \{C_i\}$ of pseudo-Boolean constraints C_i, whichever view is more convenient. A *(disjunctive) clause* $\bigvee_i \ell_i$ is equivalent to the pseudo-Boolean constraint $\sum_i \ell_i \geq 1$. Hence, formulas in *conjunctive normal form (CNF)* are special cases of pseudo-Boolean formulas.

An *assignment* is a function mapping from Boolean variables to $\{0,1\}$. *Substitutions* (or *witnesses*) generalize assignments by allowing variables to be mapped to literals, too. A substitution ω is extended to literals by $\omega(\overline{x}) = \overline{\omega(x)}$, and to preserve truth values, i.e., $\omega(0) = 0$ and $\omega(1) = 1$. For a substitution ω, the support $\mathrm{supp}(\omega)$ is the set of variables x where $\omega(x) \neq x$. A substitution α can be composed with another substitution ω by applying ω first and then α, i.e., $(\alpha \circ \omega)(x) = \alpha(\omega(x))$. Applying a substitution ω to the pseudo-Boolean constraint C in (3) yields the pseudo-Boolean constraint $C\!\upharpoonright_\omega \doteq \sum_i a_i \omega(\ell_i) \geq A$. This is extended to formulas by defining $F\!\upharpoonright_\omega \doteq \bigwedge_i C_i\!\upharpoonright_\omega$. The pseudo-Boolean constraint C is satisfied by an assignment ω if $\sum_{i:\omega(\ell_i)=1} a_i \geq A$. A pseudo-Boolean formula F is satisfied by ω if ω satisfies every constraint in F. If there is no assignment that satisfies F, then F is *unsatisfiable*.

We use the notation $F(\vec{x})$ to stress that the formula is defined over the list of variables $\vec{x} = x_1, \ldots, x_n$, where we syntactically highlight a partitioning of the list of variables by writing $F(\vec{y}, \vec{z})$ or $F(\vec{a}, \vec{b}, \vec{c})$ meaning $\vec{x} = \vec{y}, \vec{z}$ or $\vec{x} = \vec{a}, \vec{b}, \vec{c}$, respectively (denoting concatenation of the lists of variables). To apply a substitution ω element-wise to a list of literals we write $\vec{\ell}\!\upharpoonright_\omega = \omega(\ell_1), \ldots, \omega(\ell_n)$. For a formula $F(\vec{x})$ and a list of literals and truth values $\vec{y} = y_1, \ldots, y_n$, the notation $F(\vec{y})$ is syntactic sugar for $F\!\upharpoonright_\omega$ with the implicitly defined substitution $\omega(x_i) = y_i$ for $i = 1, \ldots, n$. Finally, we write $\mathrm{var}(F)$ for the set of variables in a formula F.

2.1 The VeriPB Proof System

The proof system introduced by [BGMN23] (which we will refer to as the *original system*) can prove optimal values for *optimization problems* (F, f), where F is a pseudo-Boolean formula, and f is an integer linear *objective* function over literals to be minimized subject to satisfying F. The *satisfiability (SAT) problem* is a special case by having $f = 0$ and F being a CNF formula. Proving the unsatisfiability of F then corresponds to proving that ∞ is a lower bound for (F, f). For clarity of exposition,

we focus on decision problems, i.e., problems with objective function $f = 0$, but the results can easily be extended to optimization problems as in [BGMN23].

A proof in this proof system consists of a sequence of rule applications, each deriving a new constraint. For implicational reasoning, the *cutting planes* proof system [CCT87] is used, which provides sound reasoning rules to derive pseudo-Boolean constraints implied by a pseudo-Boolean formula F, e.g., taking positive integer linear combinations or dividing by an integer and rounding up. We write $F \vdash C$ if there is a cutting planes proof deriving C from F. A set of constraints F' is derivable from another set F, denoted by $F \vdash F'$, if $F \vdash C$ for all $C \in F'$.

The proof system also has rules for deriving constraints which are not implied. To do this, the original system keeps track of two pseudo-Boolean formulas (i.e., sets of constraints), called *core* $\mathcal{C}$ and *derived* $\mathcal{D}$, which [JHB12] call irredundant and redundant clauses, respectively. In addition, we need a pseudo-Boolean formula $\mathcal{O}_{\preceq}(\vec{u}, \vec{v})$, encoding a preorder, i.e., a reflexive and transitive relation. This preorder is used to compare assignments α, β over the literals in a list $\vec{z}$ and we write $\alpha \preceq \beta$ if $\mathcal{O}_{\preceq}(\vec{z}\restriction_{\alpha}, \vec{z}\restriction_{\beta})$ evaluates to true. For a preorder $\preceq$, we define the strict order $\prec$ such that $\alpha \prec \beta$ holds if $\alpha \preceq \beta$ and $\beta \npreceq \alpha$.

We call the tuple $(\mathcal{C}, \mathcal{D}, \mathcal{O}_{\preceq}, \vec{z})$ a *configuration*. Formally, proof rules incrementally modify the configuration. To handle optimization problems, the configuration of [BGMN23] also contains the current upper bound on f, which we can omit for decision problems.

The proof system maintains two invariants: (1) $\mathcal{C}$ is satisfiable if F is satisfiable, and (2) for any assignments α satisfying $\mathcal{C}$ there exists an assignment α' satisfying $\mathcal{C}$, $\mathcal{D}$, and $\alpha' \preceq \alpha$. Starting with the configuration $(F, \emptyset, \emptyset, \emptyset)$, any valid derivation of a configuration $(\mathcal{C}, \mathcal{D}, \mathcal{O}_{\preceq}, \vec{z})$ with $\bot \in \mathcal{C} \cup \mathcal{D}$ proves that F is unsatisfiable.

Proof Rules. We list the satisfiability version of the proof rules from [BGMN23] our work modifies; all other rules in the original proof system remain unchanged.

- The *redundance-based strengthening rule* (or *redundance rule* for short) allows transitioning from $(\mathcal{C}, \mathcal{D}, \mathcal{O}_{\preceq}, \vec{z})$ to $(\mathcal{C}, \mathcal{D} \cup \{C\}, \mathcal{O}_{\preceq}, \vec{z})$ if a substitution ω and cutting planes proofs are provided showing that

$$\mathcal{C} \cup \mathcal{D} \cup \{\neg C\} \vdash (\mathcal{C} \cup \mathcal{D} \cup \{C\})\restriction_{\omega} \cup \mathcal{O}_{\preceq}(\vec{z}\restriction_{\omega}, \vec{z})\,. \tag{4}$$

- The *dominance-based strengthening rule* (or *dominance rule* for short) allows transitioning from $(\mathcal{C}, \mathcal{D}, \mathcal{O}_{\preceq}, \vec{z})$ to $(\mathcal{C}, \mathcal{D} \cup \{C\}, \mathcal{O}_{\preceq}, \vec{z})$ if a substitution ω and cutting planes proofs are provided showing that

$$\mathcal{C} \cup \mathcal{D} \cup \{\neg C\} \vdash \mathcal{C}\restriction_{\omega} \cup \mathcal{O}_{\preceq}(\vec{z}\restriction_{\omega}, \vec{z}) \tag{5}$$

$$\mathcal{C} \cup \mathcal{D} \cup \{\neg C\} \cup \mathcal{O}_{\preceq}(\vec{z}, \vec{z}\restriction_{\omega}) \vdash \bot\,. \tag{6}$$

We now briefly explain why the redundance rule preserves the second invariant. Let α be an assignment satisfying $\mathcal{C}$. Since the invariant holds for $(\mathcal{C}, \mathcal{D}, \mathcal{O}_{\preceq}, \vec{z})$, there exists an assignment α' satisfying $\mathcal{C} \cup \mathcal{D}$ and $\alpha' \preceq \alpha$. If α' happens to satisfy C, we are done. Otherwise, the derivation (4) guarantees that $\alpha' \circ \omega$ satisfies $\mathcal{C} \cup \mathcal{D} \cup \{C\}$ and $\mathcal{O}_{\preceq}(\vec{z}\restriction_{\alpha' \circ \omega}, \vec{z}\restriction_{\alpha'})$, i.e., $\alpha' \circ \omega \preceq \alpha'$. By transitivity we get $\alpha' \circ \omega \preceq \alpha$.

For the dominance rule, α' might have to be composed with ω repeatedly, but the process is guaranteed to eventually satisfy C, since the composed assignment strictly decreases with respect to the order, which is encoded by (6).

Preorders. Before using a preorder $O_\preceq$, it needs to be proven within the proof system that $O_\preceq$ is indeed reflexive and transitive. For this, the original system requires cutting planes proofs for $\emptyset \vdash O_\preceq(\vec{u}, \vec{u})$ and $O_\preceq(\vec{u}, \vec{v}) \cup O_\preceq(\vec{v}, \vec{w}) \vdash O_\preceq(\vec{u}, \vec{w})$, where $\vec{w}$ is of the same size as $\vec{u}$ and $\vec{v}$.

2.2 Symmetry Breaking

We briefly review symmetry breaking as it is used in practice, which we want to certify. Typically, symmetry breaking considers permutations σ between literals with $\sigma(\overline{\ell}) = \overline{\sigma(\ell)}$ for all literals ℓ, and finite support $\mathrm{supp}(\sigma)$. Practical symmetry breaking algorithms only detect *syntactic* symmetries of a formula F, i.e., permutations σ with $F\!\upharpoonright_\sigma \doteq F$.

To encode an ordering of assignments, typically the *lex-leader* constraint is used. Let S be a set of detected symmetries in a formula F, and $z_1, \ldots, z_n$ be variables with $\mathrm{supp}(\sigma) \subseteq \{z_1, \ldots, z_n\}$ for all $\sigma \in S$. Then we define the *lexicographic order* $\preceq_{\mathrm{lex}}$ over assignments α, β and the *lex-leader constraint* B_σ for a symmetry $\sigma \in S$ as

$$\alpha \preceq_{\mathrm{lex}} \beta \text{ iff } \sum_{i=1}^{n} 2^{n-i}\alpha(z_i) \leq \sum_{i=1}^{n} 2^{n-i}\beta(z_i) \tag{7}$$

$$B_\sigma \doteq \sum_{i=1}^{n} 2^{n-i}(\sigma(x_i) - x_i) \geq 0\,. \tag{8}$$

Intuitively, (8) constrains assignments to be smaller w.r.t. the preorder in (7) than their symmetric counterpart. Symmetry breaking introduces the constraints B_σ for $\sigma \in S$ such that if F is satisfiable, then $F \cup \bigcup_{\sigma \in S} B_\sigma$ is satisfiable.

When doing proof logging for symmetry breaking, the dominance rule in the original system can derive the lex-leader constraints B_σ as follows. Suppose that the symmetry breaker detect symmetries $\sigma_1, \ldots, \sigma_m$ of C. To log these symmetries, we use the order defined by

$$O_{\preceq_{\mathrm{lex}}}(\vec{x}, \vec{y}) = \left\{ \sum_{i=1}^{n} 2^{n-i}(y_i - x_i) \geq 0 \right\}\,. \tag{9}$$

To add a constraint B_{σ_i} to $\mathcal{D}$, we use the dominance rule with witness σ_i. The application of this rule is justified by $C \vdash C\!\upharpoonright_{\sigma_i}$ (trivial, because σ_i is a symmetry of C), and the fact that $\neg B_{\sigma_i}$ implies both $O_{\preceq_{\mathrm{lex}}}(\vec{z}\!\upharpoonright_{\sigma_i}, \vec{z})$ and $\neg O_{\preceq_{\mathrm{lex}}}(\vec{z}, \vec{z}\!\upharpoonright_{\sigma_i})$.

When using symmetry breaking for the SAT problem, the symmetry breaker instead encodes $\vec{x} \preceq_{\mathrm{lex}} \sigma(\vec{x})$ as the clauses

$$s_1 + \overline{x_1} \geq 1\,, \quad s_{i+1} + \overline{s_i} + \overline{x_{i+1}} \geq 1\,, \tag{10a}$$

$$s_1 + \sigma(x_1) \geq 1\,, \quad s_{i+1} + \overline{s_i} + \sigma(x_{i+1}) \geq 1\,, \tag{10b}$$

$$\sigma(x_1) + \overline{x_1} \geq 1\,, \quad \overline{s_i} + \sigma(x_{i+1}) + \overline{x_{i+1}} \geq 1\,, \tag{10c}$$

where s_i encodes that $(x_1, \ldots, x_i) = (\sigma(x_1), \ldots, \sigma(x_i))$. These clauses can be derived from (8) using redundance.

3 Strengthening with Auxiliary Variables

While the method presented in Section 2.2 enables proof logging for symmetry breaking, encoding the coefficients in (8) and (9) grows quadratically in the size of $\vec{z}$, which often includes all variables in the formula, making proof logging for large symmetries infeasible in practice. For proof checking, the situation is even more dire, as the proof checker has to reason internally with arbitrary-precision integer arithmetic to handle the coefficients in (8) and (9).

One way to avoid these big integers would be to represent the order as a set of clauses as in Equation (10a)–(10c), using a list of extension variables $\vec{s}$. However, this leads to challenges when defining the actual preorder $\preceq$. For an order without extension variables, we define $\alpha \preceq \beta$ to hold if $O_{\preceq}(\vec{z}\restriction_{\alpha}, \vec{z}\restriction_{\beta})$ is true. However, for a formula $O_{\preceq}(\vec{x}, \vec{y}, \vec{s})$ containing extension variables $\vec{s}$ this does not work, since the variables $\vec{s}$ in $O_{\preceq}(\vec{z}\restriction_{\alpha}, \vec{z}\restriction_{\beta}, \vec{s})$ are unassigned and in general $O_{\preceq}(\vec{z}\restriction_{\alpha}, \vec{z}\restriction_{\beta}, \vec{s})$ will not hold for all assignments to $\vec{s}$.

Instead, what we are trying to capture is that $\alpha \preceq \beta$ holds precisely when $O_{\preceq}(\vec{z}\restriction_{\alpha}, \vec{z}\restriction_{\beta}, \vec{s})$ holds, *provided that* the extension variables $\vec{s}$ are set in the right way. Equivalently, we want to say that $\alpha \preceq \beta$ holds precisely when there exists an assignment ϱ to $\vec{s}$ such that $O_{\preceq}(\vec{z}\restriction_{\alpha}, \vec{z}\restriction_{\beta}, \vec{s}\restriction_{\varrho})$ holds.

However, just adding extension variables to the proof obligations $O_{\preceq}(\vec{z}\restriction_{\omega}, \vec{z})$ in the redundance and dominance rules would not work. What we would need to show is that *some* assignment to $\vec{s}$ exists such that $O_{\preceq}(\vec{z}\restriction_{\omega}, \vec{z}, \vec{s})$ holds, but the proof system cannot express existential quantification. While the proof rules could specify the value of all extension variables $\vec{s}$, this would be very cumbersome. However, in all applications we have in mind, the preorder with extension variables already contains the information how to set the extension variables $\vec{s}$, since the extension variables are *defined* (functionally) in terms of the other variables.

To make this precise, let $\mathcal{S}_{\preceq}(\vec{x}, \vec{y}, \vec{s})$ be a *definition* of $\vec{s}$ in terms of the other variables (i.e., each assignment to the $\vec{x}$ and $\vec{y}$ can uniquely be extended to an assignment to $\vec{s}$ that satisfies $\mathcal{S}_{\preceq}(\vec{x}, \vec{y}, \vec{s})$). We now redefine $\preceq$ such that $\alpha \preceq \beta$ holds precisely when there exists an assignment ϱ to $\vec{s}$ such that $\mathcal{S}_{\preceq}(\vec{z}\restriction_{\alpha}, \vec{z}\restriction_{\beta}, \vec{s}\restriction_{\varrho}) \wedge O_{\preceq}(\vec{z}\restriction_{\alpha}, \vec{z}\restriction_{\beta}, \vec{s}\restriction_{\varrho})$ holds.

In this case, whenever we need to show that $\alpha \preceq \beta$, because of the definitional nature of $\mathcal{S}_{\preceq}$ we can *assume* that $\mathcal{S}_{\preceq}(\vec{z}\restriction_{\alpha}, \vec{z}\restriction_{\beta}, \vec{s}\restriction_{\varrho})$ holds and derive $O_{\preceq}(\vec{z}\restriction_{\alpha}, \vec{z}\restriction_{\beta}, \vec{s}\restriction_{\varrho})$ from this, thereby completely eliminating the need for providing an assignment to $\vec{s}$ in every rule application. In our actual proof system, we relax the condition that $\mathcal{S}_{\preceq}$ is definitional slightly, but intuitively, $\mathcal{S}_{\preceq}$ is best thought of as a circuit defining the value of $\vec{s}$ in terms of the other variables.

An important restriction for this to be sound is that the extension variables $\vec{s}$ in the preorder, which we call *auxiliary variables*, do not appear outside the preorder.

We now formalize this. As mentioned in Section 2.1, we focus on decision problems, and refer to Appendix B for the extension to optimization problems and proofs of all results.

3.1 Specifications

Let $\vec{a}$ be a list of variables. A pseudo-Boolean formula $\mathcal{S}(\vec{x}, \vec{a})$ is a *specification over the variables* $\vec{a}$, if it is derivable from the empty formula $\emptyset$ by the redundance rule, where each application only witnesses over variables in $\vec{a}$.

Definition 1. A formula $\mathcal{S}(\vec{x}, \vec{a}) = \{C_1, C_2, \ldots, C_n\}$ is a *specification over the variables* $\vec{a}$, if there is a list

$$(C_1, \omega_1), (C_2, \omega_2), \ldots, (C_n, \omega_n)$$

which satisfies the following:

1. The constraint C_1 can be obtained from the empty formula $\emptyset$ using the redundance rule with witness ω_1.
2. For each $i \in \{2, \ldots, n\}$ we have that C_i can be added by the redundance rule to $\bigcup_{j=1}^{i-1}\{C_j\}$ with the witness ω_i.
3. For every witness ω_i, $\operatorname{supp}(\omega_i) \subseteq \vec{a}$ holds.

A crucial property of specifications is that we can recover an assignment of the auxiliary variables from the assignment of the non-auxiliary variables. We state this property below.

Lemma 1. *Let $\mathcal{S}(\vec{x}, \vec{a})$ be a specification over $\vec{a}$. Let α be any assignment of the variables $\vec{x}$. Then, α can be extended to an assignment α', such that*

1. *α' satisfies $\mathcal{S}$, and*
2. *$\alpha(x) = \alpha'(x)$ holds for every $x \in \vec{x}$.*

To explain why Lemma 1 holds, recall from Section 2.1 that the redundance rule satisfies the following: if a constraint C is added by redundance with witness ω, then given an assignment α satisfying all other constraints, either α or $\alpha \circ \omega$ also satisfies C. Hence, defining α' by composing α with the witnesses ω_i corresponding to the constraints that do not already hold ensures that α' satisfies $\mathcal{S}$, and the fact that each witness ω_i is the identity on $\vec{x}$ (since $\operatorname{supp}(\omega_i) \subseteq \vec{a}$) ensures that $\alpha(x) = \alpha'(x)$ for $x \in \vec{x}$.

3.2 Orders with Auxiliary Variables

Next, we explain how two pseudo-Boolean formulas $O_{\preceq}(\vec{u}, \vec{v}, \vec{a})$ and $\mathcal{S}_{\preceq}(\vec{u}, \vec{v}, \vec{a})$, together with the two disjoint lists of variables $\vec{z}$ and $\vec{a}$, define a preorder.

Definition 2. Let $\vec{u}$ and $\vec{v}$ be disjoint lists of variables of size n and let $\vec{a}$ be a list of auxiliary variables. Let $O_{\preceq}(\vec{u}, \vec{v}, \vec{a})$ and $\mathcal{S}_{\preceq}(\vec{u}, \vec{v}, \vec{a})$ be two pseudo-Boolean formulas such that $\mathcal{S}_{\preceq}$ is a specification over $\vec{a}$.

Then we define the relation $\preceq$ over the domain of total assignments to a list of variables $\vec{z}$ of size n as follows: For assignments α, β we let $\alpha \preceq \beta$ hold, if and only if there exists an assignment ϱ to the variables $\vec{a}$, such that

$$\mathcal{S}_{\preceq}(\vec{z}\restriction_{\alpha}, \vec{z}\restriction_{\beta}, \vec{a}\restriction_{\varrho}) \wedge O_{\preceq}(\vec{z}\restriction_{\alpha}, \vec{z}\restriction_{\beta}, \vec{a}\restriction_{\varrho})$$

evaluates to true.

To ensure that $\mathcal{O}_\preceq$ and $\mathcal{S}_\preceq$ actually define a preorder, we require cutting planes proofs that show reflexivity, i.e., $\emptyset \vdash \alpha \preceq \alpha$, and transitivity, i.e., $\alpha \preceq \beta \wedge \beta \preceq \gamma \vdash \alpha \preceq \gamma$. To write these proof obligations using the cutting planes proof system, which cannot handle an existentially quantified conclusion, we can use the specification as a premise. The specification premise essentially tells us which auxiliary variables the existential quantifier should pick. In particular, for reflexivity, the proof obligation is

$$\mathcal{S}_\preceq(\vec{x},\vec{x},\vec{a}) \vdash \mathcal{O}_\preceq(\vec{x},\vec{x},\vec{a})\,. \tag{11}$$

For transitivity, the proof obligation is

$$\begin{aligned}\mathcal{S}_\preceq(\vec{x},\vec{y},\vec{a}) \cup \mathcal{O}_\preceq(\vec{x},\vec{y},\vec{a}) \cup \mathcal{S}_\preceq(\vec{y},\vec{z},\vec{b}) \\ \cup\, \mathcal{O}_\preceq(\vec{y},\vec{z},\vec{b}) \cup \mathcal{S}_\preceq(\vec{x},\vec{z},\vec{c}) \vdash \mathcal{O}_\preceq(\vec{x},\vec{z},\vec{c})\,.\end{aligned} \tag{12}$$

Intuitively, (12) says that if the circuits defining the auxiliary variables are correctly evaluated, which is encoded by the premises $\mathcal{S}_\preceq(\vec{x},\vec{y},\vec{a}) \cup \mathcal{S}_\preceq(\vec{y},\vec{z},\vec{b}) \cup \mathcal{S}_\preceq(\vec{x},\vec{z},\vec{c})$, then transitivity should hold, i.e., $\mathcal{O}_\preceq(\vec{x},\vec{y},\vec{a}) \cup \mathcal{O}_\preceq(\vec{y},\vec{z},\vec{b}) \vdash \mathcal{O}_\preceq(\vec{x},\vec{z},\vec{c})$. However, if the auxiliary variables are not correctly set, then no claims are made.

These proof obligations ensure that $\preceq$ is a preorder:

Lemma 2. *If $\mathcal{O}_\preceq$ and $\mathcal{S}_\preceq$ satisfy Equations* (11) *and* (12)*, then $\preceq$ as defined by $\mathcal{O}_\preceq$ and $\mathcal{S}_\preceq$ is a preorder.*

3.3 Validity

We extend the configurations of the proof system to $(\mathcal{C},\mathcal{D},\mathcal{O}_\preceq,\mathcal{S}_\preceq,\vec{z},\vec{a})$. In particular, we extend the notion of *weak-(F,f)-validity* from [BGMN23] to our new configurations, focusing on decision problems:

Definition 3. A configuration $(\mathcal{C},\mathcal{D},\mathcal{O}_\preceq,\mathcal{S}_\preceq,\vec{z},\vec{a})$ is *weakly F-valid* if the following conditions hold:

1. If F is satisfiable, then $\mathcal{C}$ is satisfiable.
2. For every assignment α satisfying $\mathcal{C}$, there exists an assignment α' satisfying $\mathcal{C} \cup \mathcal{D}$ and $\alpha' \preceq \alpha$.

Furthermore, we assume that for any configuration $(\mathcal{C},\mathcal{D},\mathcal{O}_\preceq,\mathcal{S}_\preceq,\vec{z},\vec{a})$ the following conditions hold:

1. $\mathcal{O}_\preceq$ and $\mathcal{S}_\preceq$ refer to formulas for which Equations (11) and (12) have been successfully proven.
2. $\mathcal{S}_\preceq$ is a specification over $\vec{a}$.
3. The variables $\vec{a}$ only occur in $\mathcal{O}_\preceq$ and $\mathcal{S}_\preceq$, and these variables are disjoint from $\vec{z}$.

Observe that due to these invariants, satisfying assignments for $\mathcal{C} \cup \mathcal{D}$ do not need to assign the variables $\vec{a}$. In the following, we assume that such assignments are indeed defined only over the domain var$(\mathcal{C} \cup \mathcal{D})$.

3.4 Dominance-Based Strengthening Rule

As in the original system, the dominance rule allows adding a constraint C to the derived set using witness ω if from the premises $\mathcal{C} \cup \mathcal{D} \cup \{\neg C\}$ we can derive $\mathcal{C}\restriction_\omega$ and show that $\alpha \circ \omega \prec \alpha$ holds for all assignments α satisfying $\mathcal{C} \cup \mathcal{D} \cup \{\neg C\}$. To show $\alpha \circ \omega \prec \alpha$, we separately show that $\alpha \circ \omega \preceq \alpha$ and $\alpha \npreceq \alpha \circ \omega$. To show that $\alpha \circ \omega \preceq \alpha$, we have to show that $\mathcal{O}_\preceq(\vec{z}\restriction_\omega, \vec{z}, \vec{a})$, assuming that the circuit defining the auxiliary variables $\vec{a}$ has been evaluated correctly, which is encoded by the specification $\mathcal{S}_\preceq(\vec{z}\restriction_\omega, \vec{z}, \vec{a})$. This leads to the proof obligation

$$\mathcal{C} \cup \mathcal{D} \cup \{\neg C\} \cup \mathcal{S}_\preceq(\vec{z}\restriction_\omega, \vec{z}, \vec{a}) \vdash \mathcal{O}_\preceq(\vec{z}\restriction_\omega, \vec{z}, \vec{a})\,. \tag{13}$$

To show that $\alpha \npreceq \alpha \circ \omega$, we have to show that $\neg\mathcal{O}_\preceq(\vec{z}, \vec{z}\restriction_\omega, \vec{a})$ assuming $\mathcal{S}_\preceq(\vec{z}, \vec{z}\restriction_\omega, \vec{a})$. However, since $\neg\mathcal{O}_\preceq(\vec{z}, \vec{z}\restriction_\omega, \vec{a})$ is not necessarily a pseudo-Boolean formula (due to the negation), we instead show that we can derive contradiction from $\mathcal{O}_\preceq(\vec{z}, \vec{z}\restriction_\omega, \vec{a})$, leading to the proof obligation:

$$\mathcal{C} \cup \mathcal{D} \cup \{\neg C\} \cup \mathcal{S}_\preceq(\vec{z}, \vec{z}\restriction_\omega, \vec{a}) \cup \mathcal{O}_\preceq(\vec{z}, \vec{z}\restriction_\omega, \vec{a}) \vdash \bot\,. \tag{14}$$

The following lemma shows that these proof obligations indeed imply $\alpha \circ \omega \preceq \alpha$ and $\alpha \npreceq \alpha \circ \omega$, respectively:

Lemma 3. *Let G be a formula and ω a witness with* $\operatorname{supp}(\omega) \subseteq \operatorname{var}(G)$. *Furthermore, let $\vec{a} \cap \operatorname{var}(G) = \emptyset$ and $\mathcal{S}_\preceq$ be a specification over $\vec{a}$. Also, let $\mathcal{O}_\preceq$ and $\mathcal{S}_\preceq$ define a pre-order $\preceq$. Then the following hold:*

1. *If $G \cup \mathcal{S}_\preceq(\vec{z}\restriction_\omega, \vec{z}, \vec{a}) \vdash \mathcal{O}_\preceq(\vec{z}\restriction_\omega, \vec{z}, \vec{a})$ holds, then for each assignment α satisfying G, $\alpha \circ \omega \preceq \alpha$ holds.*
2. *If $G \cup \mathcal{S}_\preceq(\vec{z}, \vec{z}\restriction_\omega, \vec{a}) \cup \mathcal{O}_\preceq(\vec{z}, \vec{z}\restriction_\omega, \vec{a}) \vdash \bot$ holds, then for each assignment α satisfying G, $\alpha \npreceq \alpha \circ \omega$ holds.*

Hence, we define the dominance rule as follows:

Definition 4 (Dominance-based strengthening with specification)**.** We can transition from the configuration $(\mathcal{C}, \mathcal{D}, \mathcal{O}_\preceq, \mathcal{S}_\preceq, \vec{z}, \vec{a})$ to $(\mathcal{C}, \mathcal{D} \cup \{C\}, \mathcal{O}_\preceq, \mathcal{S}_\preceq, \vec{z}, \vec{a})$ using the dominance rule if the following conditions are met:

1. The constraint C does not contain variables in $\vec{a}$.
2. There is a witness ω for which $\operatorname{image}(\omega) \cap \vec{a} = \emptyset$ holds.
3. We have cutting planes proofs for the following:

$$\mathcal{C} \cup \mathcal{D} \cup \{\neg C\} \cup \mathcal{S}_\preceq(\vec{z}\restriction_\omega, \vec{z}, \vec{a}) \vdash \mathcal{C}\restriction_\omega \cup \mathcal{O}_\preceq(\vec{z}\restriction_\omega, \vec{z}, \vec{a}) \tag{15}$$

$$\mathcal{C} \cup \mathcal{D} \cup \{\neg C\} \cup \mathcal{S}_\preceq(\vec{z}, \vec{z}\restriction_\omega, \vec{a}) \cup \mathcal{O}_\preceq(\vec{z}, \vec{z}\restriction_\omega, \vec{a}) \vdash \bot\,. \tag{16}$$

Using Lemma 3, we can show that the dominance rule preserves the invariants required by weak F-validity:

Lemma 4. *If we can transition from $(\mathcal{C}, \mathcal{D}, \mathcal{O}_\preceq, \mathcal{S}_\preceq, \vec{z}, \vec{a})$ to $(\mathcal{C}, \mathcal{D} \cup \{C\}, \mathcal{O}_\preceq, \mathcal{S}_\preceq, \vec{z}, \vec{a})$ by the dominance rule, and $(\mathcal{C}, \mathcal{D}, \mathcal{O}_\preceq, \mathcal{S}_\preceq, \vec{z}, \vec{a})$ is weakly F-valid, then the configuration $(\mathcal{C}, \mathcal{D} \cup \{C\}, \mathcal{O}_\preceq, \mathcal{S}_\preceq, \vec{z}, \vec{a})$ is also weakly F-valid.*

3.5 Redundance-Based Strengthening Rule

We also modify the redundance rule to work in our extended proof system. Similarly to the dominance rule, we can use $\mathcal{S}_{\preceq}(\vec{z}\restriction_\omega, \vec{z}, \vec{a})$ as an extra premise in our proof obligations.

Definition 5 (Redundance-based strengthening with specification)**.** We can transition from the configuration $(\mathcal{C}, \mathcal{D}, \mathcal{O}_{\preceq}, \mathcal{S}_{\preceq}, \vec{z}, \vec{a})$ to $(\mathcal{C}, \mathcal{D} \cup \{C\}, \mathcal{O}_{\preceq}, \mathcal{S}_{\preceq}, \vec{z}, \vec{a})$ using the redundance rule if the following conditions are met:

1. The constraint C does not contain variables in $\vec{a}$.

2. There is a witness ω for which $\mathrm{image}(\omega) \cap \vec{a} = \emptyset$ holds.

3. We have cutting planes proof that the following holds:

$$\begin{aligned} &\mathcal{C} \cup \mathcal{D} \cup \{\neg C\} \cup \mathcal{S}_{\preceq}(\vec{z}\restriction_\omega, \vec{z}, \vec{a}) \\ &\quad \vdash (\mathcal{C} \cup \mathcal{D} \cup \{C\})\restriction_\omega \cup \mathcal{O}_{\preceq}(\vec{z}\restriction_\omega, \vec{z}, \vec{a})\,. \end{aligned} \tag{17}$$

The redundance rule preserves weak F-validity:

Lemma 5. *If we can transition from $(\mathcal{C}, \mathcal{D}, \mathcal{O}_{\preceq}, \mathcal{S}_{\preceq}, \vec{z}, \vec{a})$ to $(\mathcal{C}, \mathcal{D} \cup \{C\}, \mathcal{O}_{\preceq}, \mathcal{S}_{\preceq}, \vec{z}, \vec{a})$ by the redundance rule, and $(\mathcal{C}, \mathcal{D}, \mathcal{O}_{\preceq}, \mathcal{S}_{\preceq}, \vec{z}, \vec{a})$ is weakly F-valid, then the configuration $(\mathcal{C}, \mathcal{D} \cup \{C\}, \mathcal{O}_{\preceq}, \mathcal{S}_{\preceq}, \vec{z}, \vec{a})$ is also weakly F-valid.*

4 Efficient Proof Logging in satsuma

Using our extended proof system, we implement proof logging in the state-of-the-art symmetry breaker SATSUMA[1]. The details can be found in Appendix C and a worked-out example is given in Appendix D. Like in the original system (as explained in Section 2.2), the (negation of the) symmetry breaking constraints can be used to show the proof obligations for the order. However, in the extended system this is more complicated, because we need to relate two different sets of extension variables (those in the symmetry breaking constraints and the auxiliary variables in the specification).

Our new method achieves an asymptotic speedup over the old method. Defining the lexicographical order over n variables can be done in time $\mathrm{O}(n)$ with our new method (for both checking and logging), while the old method requires time $\mathrm{O}(n^2)$. Breaking a symmetry σ over $k = |\mathrm{supp}(\sigma)|$ variables ($k \leq n$) takes time $\mathrm{O}(k)$ for logging and $\mathrm{O}(n)$ for checking with our new method, while the old method requires time $\mathrm{O}(nk)$ for logging and time $\mathrm{O}(n^2 + nk^2)$ for checking. Therefore, the new method is in each case asymptotically at least a factor n faster for both logging and checking than the old method used by [BGMN23].

[1] SATSUMA code: `https://doi.org/10.5281/zenodo.17607863`

5 Proof Checker Implementation

We implemented checking for our extended proof system in the proof checker VeriPB[2] and the formally verified proof checker CakePB[3]. Several optimizations are necessary to handle orders with many specification constraints efficiently.

Lazy Constraint Loading and Evaluation. When checking cutting planes derivations for the dominance or redundance rule (15)–(17), the proof checkers load the specification constraints from $\mathcal{S}_{\preceq}$ only when they are used in the proof. More specifically, the constraints in $\mathcal{S}_{\preceq}$ are not even computed until loaded explicitly, which improves the checking performance by a linear factor if the specification $\mathcal{S}_{\preceq}$ is not required for a cutting planes derivation.

Implicit Reflexivity Proof. Since the loaded order is always proven to be reflexive (11), the cutting planes derivation for $O_{\preceq}(\vec{z}\upharpoonright_{\omega}, \vec{z}, \vec{a})$ can be skipped for the redundance rule (17) if the domain of the witness ω does not contain a variable in $\vec{z}$. Requiring an explicit cutting planes derivation for $O_{\preceq}(\vec{z}\upharpoonright_{\omega}, \vec{z}, \vec{a})$ would again involve computation over all constraints in the specification $\mathcal{S}_{\preceq}$, which incurs a linear overhead for proof logging and checking.

Formal Verification. We updated CakePB in two phases, yielding the same end-to-end verification guarantees for proof checking as discussed in [GMM+24]. First, we formally verified soundness of all updates to the proof system (including Lemmas 1–5). Second, we implemented and verified these changes in the CakePB codebase, including soundness for the optimizations described above.

6 Experimental Evaluation

From the analysis in Section 4, we know that our new proof system is asymptotically better in theory. We now show that it indeed enables much faster proof logging and checking in practice, on both crafted and real-world problem instances. The experiments in this section are performed on machines with dual AMD EPYC 7643 processors, 2TBytes of RAM, and local solid state hard drives, running Ubuntu 22.04.2. We limit each individual process to 32GBytes RAM, and run up to 16 processes in parallel (having checked that this does not make a measurable difference to runtimes). We give satsumaa time limit of 1,000s per instance, and VeriPB and CakePB 10,000s. We remark that runtimes involving writing proofs are often bound by disk I/O performance; nevertheless, our general experimental trends are valid. In each case, when we run VeriPB, we run it in elaboration mode. This means that, in addition to checking a proof, it also outputs a simplified proof that is suitable for giving to CakePB. This also means that any instance that fails for VeriPB due to limits cannot be run through CakePB.

[2] VeriPB code: `https://doi.org/10.5281/zenodo.17608873`
[3] CakePB code: `https://doi.org/10.5281/zenodo.17609070`

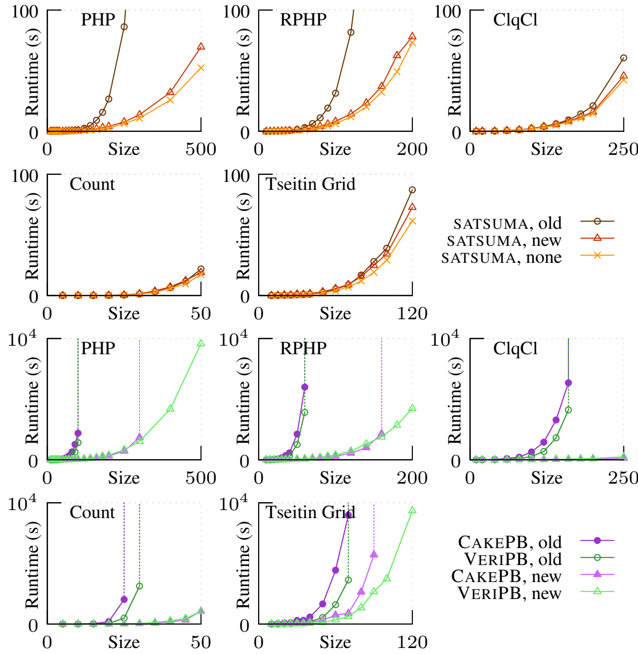


Figure 1: *On top, the cost of running* SATSUMA*with or without proof logging, on crafted benchmark instances, as the instance size grows. In each case logging with the new method scales similarly to not doing logging, whilst the old method exhibits worse scaling for several families. On the bottom, the cost of checking these proofs: the new method exhibits better scaling.*

The aim of our experiments is not to determine whether symmetry breaking is a good idea, or how it should be done. Indeed, SATSUMAproduces the same CNF (modulo a potential sorting of the constraints) regardless of whether it is outputting proofs using the old method or the new method, or not outputting proofs at all. Thus, we limit our experiments to checking that the proofs produced by SATSUMAare in fact valid, rather than reporting times for checking the entire solving process. This allows us to precisely measure the effects of our changes.

We perform experiments across two sets of instances, with different purposes. Our first set consists of five families of crafted benchmarks which have well-understood symmetries, generated using CNFGEN [LENV17]. We note that the number of variables grows quadratically or cubically in the instance size. Appendix E provides more details about the crafted benchmarks.

We show the results in Figure 1. For each of the five families, on the top row we plot the time needed to run SATSUMAto produce symmetry breaking constraints, without proof logging and with both kinds of proof logging enabled. In each case, the new method scales similarly to not doing proof logging, although there is a cost to be paid to output the proofs to disk. However, particularly for the PHP and RPHP families, it is clear that even writing the proofs is both asymptotically

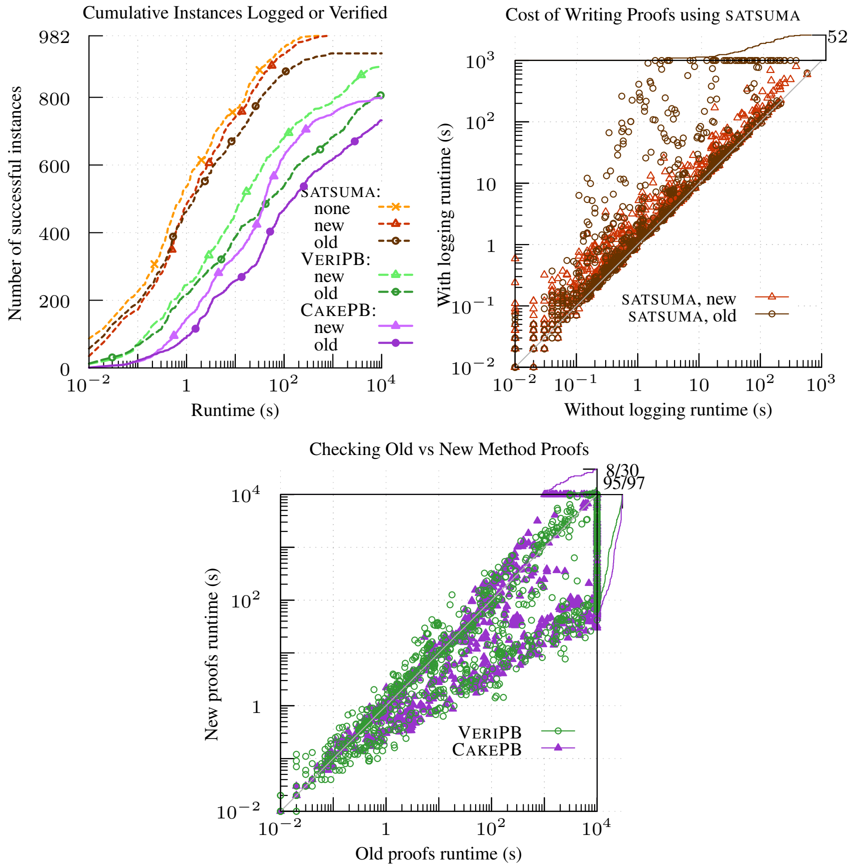


Figure 2: *On the left, the cumulative number of "interesting" SAT competition instances broken, logged, and checked over time. The centre plot compares the added cost of writing proofs with the old and new methods, compared to not writing proofs; 52 instances reached time or memory limits with the old method. The right-hand plot compares the cost of checking proofs using the old and new methods, with points below the diagonal line showing where both methods succeeded but the new method was faster; additionally, 8 and 30 instances reach limits with VeriPB and CakePB respectively with the new method where the old method succeeded, compared to 95 and 97 respectively with the old method where the new method succeeded.*

and practically much more expensive using the old method. On the bottom row of the figure, we plot the checking times. We see much better scaling from the new method in all five cases. Formally verified proof checking using CakePB is slightly slower than with VeriPB, which is not surprising—for the families where CakePB's curve stops on smaller instances, this is due to CakePB hitting memory limits when VeriPB did not. The new method is particularly helpful for CakePB

as its formally verified arbitrary precision arithmetic library is known to be less efficient than other (unverified) libraries [TMK+19].

Our second set of instances are taken from the SAT competition [IJ24]. Because we are only interested in instances where we can measure something interesting about symmetry breaking, we selected the 982 instances from the main competition tracks from 2020 to 2024 where SATSUMAwas able to run to completion and identify at least one symmetry. In the left-hand plot of Figure 2, we show the cumulative number of instances successfully logged or checked over time. The leftmost ("best") curve is to run SATSUMAwith no proof logging, and this is closely followed by running SATSUMAwith proof logging using the new method, where we could produce proofs for all 982 instances. When producing proofs using the old method, in contrast, we were only able to produce proofs for 930 instances before limits were reached. We were able to check the correctness of the symmetry breaking constraints for the new method for 893 instances (799 with CAKEPB), and with the old method for only 806 instances (732 with CAKEPB). The two scatter plots in the figure give a more detailed comparison of the added cost of running SATSUMAwith proof-logging enabled, and comparing the checking costs of the old and new proof methods. In both cases it is clear that the new method is never more than a small constant factor worse than the old method, and that it is often many orders of magnitude faster.

7 Concluding Remarks

We have presented a substantial redesign of the VERIPB proof system [BGMN23, GN21, Goc22] in order to support faster certified symmetry breaking. Central to our redesign is support for using auxiliary variables to encode ordering constraints over assignments. Theoretically, the use of orders with auxiliary variables allows us to avoid encoding lexicographical orders using big integers, that are prohibitive for problems with large symmetries; this improves on the previous state-of-the-art proof logging approach [BGMN23] by at least a linear factor. To evaluate this in practice, we implemented proof logging using our new method in the state-of-the-art symmetry breaking tool SATSUMA [ABR24], and proof checking for our extended system in VERIPB and the formally verified checker CAKEPB [GMM+24]; our experimental evaluation shows orders-of-magnitude improvement for proof logging and checking compared to the old approach.

Although proof logging is now asymptotically as fast as symmetry breaking, enabling proof logging can still incur a constant factor overhead. However, improving this would be mostly an engineering effort—we are already able to produce proofs for all of our benchmark instances within reasonable time. Checking the proof can still be asymptotically slower than symmetry breaking in theory, which leaves room to significantly improve the performance of proof checking for symmetry breaking. The key challenge here is that the proof checker currently has to reason about all variables in the order for each symmetry broken, while the symmetry breaker does this once at a higher level. Future work could investigate proof logging for conditional and dynamic symmetry breaking during

search [GKL+05] or other dynamic methods of exploiting symmetries [DBB17], in contrast to the static symmetry breaking we presented here.

Acknowledgments

We would like to thank anonymous reviewers of *Pragmatics of SAT* and *AAAI* for their useful comments.

This work is partially funded by the European Union (ERC, CertiFOX, 101122653). Views and opinions expressed are however those of the author(s) only and do not necessarily reflect those of the European Union or the European Research Council. Neither the European Union nor the granting authority can be held responsible for them.

This work is also partially funded by the Fonds Wetenschappelijk Onderzoek – Vlaanderen (project G064925N).

Benjamin Bogø and Jakob Nordström are funded by the Independent Research Fund Denmark grant 9040-00389B. Jakob Nordström is also funded by the Swedish Research Council grant 2024-05801. Wietze Koops and Andy Oertel are funded by the Wallenberg AI, Autonomous Systems and Software Program (WASP) funded by the Knut and Alice Wallenberg Foundation. Benjamin Bogø, Wietze Koops, Jakob Nordström and Andy Oertel also acknowledge to have benefited greatly from being part of the Basic Algorithms Research Copenhagen (BARC) environment financed by the Villum Investigator grant 54451.

Ciaran McCreesh and Arthur Gontier were supported by the Engineering and Physical Sciences Research Council grant number EP/X030032/1. Ciaran McCreesh was also supported by a Royal Academy of Engineering research fellowship.

Magnus Myreen is funded by the Swedish Research Council grant 2021-05165.

Adrian Rebola-Pardo is funded in whole or in part by the Austrian Science Fund (FWF) BILAI project 10.55776/COE12.

Yong Kiam Tan was supported by the Singapore NRF Fellowship Programme NRF-NRFF16-2024-0002.

Our computational experiments used resources provided by LUNARC at Lund University.

Different subsets of the authors wish to thank the participants of the Dagstuhl workshops 22411 *Theory and Practice of SAT and Combinatorial Solving* and 25231 *Certifying Algorithms for Automated Reasoning* and of the *1st International Workshop on Highlights in Organizing and Optimizing Proof-logging Systems (WHOOPS '24)* at the University of Copenhagen for several stimulating discussions.

Appendix A The Cutting Planes Proof System

In this appendix, the cutting planes proof system is explained in sufficient detail to understand the rest of the appendix. We will also present the syntax in the VeriPB format used to write the proofs to a file and for the example in Appendix D. The cutting planes proof system was first introduced by [CCT87] and more details about recent results about cutting planes can be found in [BN21].

As mentioned in Section 2, a *Boolean variable* x can take values 0 (false) and 1 (true). A *literal* is either the variable x itself or its negation $\overline{x} = 1 - x$. A *pseudo-Boolean constraint* is an integer linear inequality over literals ℓ_i

$$C \doteq \sum_i a_i \ell_i \geq A\,, \tag{18}$$

where without loss of generality the coefficients a_i and the right-hand side A are non-negative and the literals ℓ_i are over distinct variables. A (partial) assignment is a (partial) function mapping variables to 0 and 1, which is extended to literals in the natural way. The *cutting planes* proof system provides rules to derive a constraint C that is implied by a formula F, i.e., F and $F \cup \{C\}$ have the same set of satisfying assignments.

In the VeriPB syntax a constraint is referred to by a positive integer called its *constraint ID*. The constraint IDs are consecutively assigned to the constraints in the order in which they are derived and start with the original formula F getting IDs $1, \ldots, |F|$. A negative constraint ID can be used as a relative reference, e.g., the ID `-1` means the previously derived constraint. A cutting planes step starts with the keyword `pol` and ends with a semicolon. After each cutting planes step, the derived constraint is assigned the next ID and added to the derived constraints in the checker. The derivation is written in reverse Polish notation (or postfix notation), hence the checker maintains a stack of operands. Each of the following rule pops it operands from the stack and pushed on the stack its result.

Let $C \doteq \sum_i a_i \ell_i \geq A$ and $D \doteq \sum_i b_i \ell_i \geq B$ be constraints from the formula F or derived in previous steps. The constraint C can always be derived by the *constraint axiom* rule, which is just the ID of the constraint in the syntax. *Literal axioms* $x_1 \geq 0$ and $\overline{x}_1 \geq 0$ are derivable by using the syntax `x1` and `~x1`, respectively. The constraint C can be *multiplied* by an integer m by multiplying each coefficient and the right-hand side by m, resulting in $\sum_i (m a_i) \ell_i \geq mA$. This is denoted by * in the syntax and assumes the last operand on the stack is an integer and the second to last is a constraint. The constraints C and D can be *added* to derive $\sum_i (a_i + b_i) \ell_i \geq A + B$, which is denoted by + in the syntax, which assumes that the last two operands on the stack are constraints. The constraint C can be *saturated* to derive $\sum_i \min\{a_i, A\} \ell_i \geq A$, which is denoted by `s` in the syntax and expects that the last operand on the stack is a constraint. *Weakening* is syntactic sugar for adding literal axioms to the constraint C to eliminate the term $a_j \ell_j$ in C resulting $\sum_{i \neq j} a_i \ell_i \geq A - a_j$. This is denoted in the syntax by `w`, which assumes that the last operand on the stack is the variable over which ℓ is over and the second to last operand is a constraint.

VeriPB also supports the cutting planes rule of *division*, which we will mention for completeness, but we will not need it for the example in Appendix D. Considering the constraint C and a positive integer d, each coefficient and the right-hand side is divided by d and rounded up, resulting in $\sum_i \lceil a_i/d \rceil \ell_i \geq \lceil A/d \rceil$. This is denoted by `d` in the syntax and assumes that the last operand on the stack is an integer and the second to last is a constraint.

To better illustrate how the syntax relates to rules, suppose we have the example constraint

$$2\overline{x}_1 + 3x_2 + 2x_3 \geq 5\,, \tag{19}$$

with constraint ID 19. The cutting planes proof line

```
pol 19 x2 2 * + x1 w s 2 d ;
```

first pushes the constraint with ID 19 to the stack, then the literal axiom $x_2 \geq 0$ is pushed on the stack, which is immediately afterwards multiplied by 2 resulting on $2x_2 \geq 0$ being pushed on the stack. Then the last two constraints on the stack, which are $2\overline{x}_1 + 3x_2 + 2x_3 \geq 5$ and $2x_2 \geq 0$, are added, resulting in $2\overline{x}_1 + 5x_2 + 2x_3 \geq 5$. This constraint is then weakened on the variable x_1 resulting in the constraint $5x_2 + 2x_3 \geq 3$, which is saturated to yield $3x_2 + 2x_3 \geq 3$. Finally, dividing this constraint by 2 yields $2x_2 + x_3 \geq 2$, which is the result of this step and added to the derived constraints.

The *slack* of the pseudo-Boolean constraint (18) with respect to the (partial) assignment ρ is

$$slack(\sum_i a_i\ell_i \geq A; \rho) \coloneqq \sum_{i:\rho(\ell_i)\neq 0} a_i - A\,.$$

If the slack is negative, then the constraint can not be satisfied by any extension of ρ, as there are not enough literals assigned to 1 or unassigned. If there is a literal ℓ_i in the constraint C that is unassigned and $0 \leq a_i < slack(C; \rho)$, then ℓ_i has to be assigned to 1, as assigning ℓ_i to 0 would lead to a negative slack. This assignment is then added to ρ, and we say that C *propagated* ℓ_i to 1 under ρ. This process is repeated until there are no further propagations by any constraint or a constraint is falsified, i.e., negative slack. We refer to the latter case as a *conflict*.

The negation of the pseudo-Boolean constraint C in (18) is $\neg C \doteq \sum_i a_i\overline{\ell}_i \geq \sum_i a_i - A + 1$. A formula F implies a constraint C by *reverse unit propagation (RUP)* [GN03] if $F \cup \{\neg C\}$ propagtes to a conflict. This shows that $\neg C$ is falsified by all satisfying assignments to F, thus any assignment satisfying F also satisfies $F \cup \{C\}$.

In the VeriPB syntax, a RUP step contains the constraint to be derived in a modified OPB format [RM16]. Additionally, the syntax also allows for hints similar to LRAT [CHH+17] that specify which constraints have to be propagated to reach a conflict. E.g.,, the RUP step

```
rup 2 x2 1 x3 >= 2 : 19 ;
```

derives the constraint $2x_2 + x_3 \geq 2$ with the hint that only the constraint with ID 19 above ($2\overline{x}_1 + 3x_2 + 2x_3 \geq 5$) is required in addition to the negation ($2\overline{x}_2 + \overline{x}_3 \geq 2$) of the constraint to propagate from the empty assignment to a conflict.

Appendix B Full Description of Extended Proof System

In this appendix, we provide further details on the extended proof system introduced in Sections 3.1 up to 3.5, including the statement of all results in the more general context of optimization problems, and all proofs. For a discussion on the intuition behind the proof system, we refer the start of Section 3. To make it easier to read the appendix as its own section, we also include all content that was already in Sections 3.1 up to 3.5 in the main paper.

B.1 Specifications

Let $\vec{a}$ be a list of variables. A pseudo-Boolean formula $\mathcal{S}(\vec{x}, \vec{a})$ is a *specification over the variables* $\vec{a}$, if it is derivable from the empty formula $\emptyset$ by the redundance rule, where each application only witnesses over variables in $\vec{a}$.

Definition 6. A formula $\mathcal{S}(\vec{x}, \vec{a}) = \{C_1, C_2, \ldots, C_n\}$ is a *specification over the variables* $\vec{a}$, if there is a list

$$(C_1, \omega_1), (C_2, \omega_2), \ldots, (C_n, \omega_n)$$

which satisfies the following:

1. The constraint C_1 can be obtained from the empty formula $\emptyset$ using the redundance rule with witness ω_1.

2. For each $i \in \{2, \ldots, n\}$ we have that C_i can be added by the redundance rule to $\bigcup_{j=1}^{i-1}\{C_j\}$ with the witness ω_i. In other words, it should hold that

$$\bigcup_{j=1}^{i-1}\{C_j\} \cup \{\neg C_i\} \vdash \bigcup_{j=1}^{i}\{C_j{\upharpoonright}_{\omega_i}\}. \tag{20}$$

3. For every witness ω_i, $\mathrm{supp}(\omega_i) \subseteq \vec{a}$ holds.

In terms of configurations, we can say that a formula $\mathcal{S}(\vec{x}, \vec{a}) = \{C_1, C_2, \ldots, C_n\}$ is a specification over $\vec{a}$ if we can transition from the configuration $(\emptyset, \emptyset, \emptyset, \emptyset)$ to the configuration $(\emptyset, \mathcal{S}(\vec{x}, \vec{a}), \emptyset, \emptyset)$ using only the redundance rule with witnesses ω_i such that $\mathrm{supp}(\omega_i) \subseteq \vec{a}$ holds.

Remark 1. Note that the definition of the specification uses the redundance rule, while the redundance rule in our extended proof system is only defined later. This is not a problem since the applications of the redundance rule in a specification require that the loaded order is the trivial preorder $\mathcal{O}_\top \doteq \emptyset$ relating all assignments, for which it is irrelevant whether we use auxiliary variables or not.

A crucial property of specifications is that we can recover an assignment of the auxiliary variables from the assignment of the non-auxiliary variables. We state this property below.

Lemma 6. *Let $\mathcal{S}(\vec{x}, \vec{a})$ be a specification over $\vec{a}$. Let α be any assignment of the variables $\vec{x}$. Then, α can be extended to an assignment α', such that*

1. *α' satisfies $\mathcal{S}$, and*

2. *$\alpha(x) = \alpha'(x)$ holds for every $x \in \vec{x}$.*

To explain why Lemma 6 holds, recall from Section 2.1 that the redundance rule satisfies the following: if a constraint C is added by redundance with witness ω, then given an assignment α satisfying all other constraints, either α or $\alpha \circ \omega$ also satisfies C. Hence, defining α' by composing α with the witnesses ω_i corresponding to the constraints that do not already hold ensures that α' satisfies $\mathcal{S}$, and the fact that each witness ω_i is the identity on $\vec{x}$ (since $\mathrm{supp}(\omega_i) \subseteq \vec{a}$) ensures that $\alpha(x) = \alpha'(x)$ for $x \in \vec{x}$.

We now give a complete proof of Lemma 6.

Proof of Lemma 6. Write $\mathcal{S}(\vec{x}, \vec{a}) = \{C_1, C_2, \ldots, C_n\}$. We show by induction on $i \geq 0$ that we can extend α to an assignment α_i such that

1. α_i satisfies $F_i = \{C_1, C_2, \ldots, C_i\}$, and
2. $\alpha(x) = \alpha_i(x)$ holds for every $x \in \vec{x}$.

For the base case $i = 0$, we extend α to an assignment α_0 by setting all variables of $\vec{a}$ to false. Then α_0 trivially satisfies $F_0 = \emptyset$, and $\alpha(x) = \alpha_0(x)$ for every $x \in \vec{x}$ since we only change the assignment of $\vec{a}$.

We proceed with the induction step. By the induction hypothesis, we can extend α to an assignment α_i such that α_i satisfies F_i and $\alpha(x) = \alpha_i(x)$ holds for every $x \in \vec{x}$.

If α_i also happens to satisfy C_{i+1}, we set $\alpha_{i+1} = \alpha_i$. Then α_{i+1} indeed satisfies $F_{i+1} = F_i \cup \{C_{i+1}\}$ and we have $\alpha(x) = \alpha_i(x) = \alpha_{i+1}(x)$ for every $x \in \vec{x}$.

If α_i does not satisfy C_{i+1}, we set $\alpha_{i+1} = \alpha_i \circ \omega_{i+1}$. Since then α_i satisfies $\bigcup_{j=1}^{i}\{C_j\} \cup \{\neg C_{i+1}\}$, Equation (20) then implies that α_i satisfies $F_{i+1}\restriction_{\omega_{i+1}} = \bigcup_{j=1}^{i+1}\{C_j \restriction_{\omega_{i+1}}\}$, which in turn implies that $\alpha_{i+1} = \alpha_i \circ \omega_{i+1}$ satisfies F_{i+1}. Since $\mathrm{supp}(\omega_{i+1}) \subseteq \vec{a}$, we have $\omega_{i+1}(x) = x$ for every $x \in \vec{x}$, so $\alpha(x) = \alpha_i(x) = \alpha_i(\omega_{i+1}(x)) = (\alpha_i \circ \omega_{i+1})(x) = \alpha_{i+1}(x)$ for every $x \in \vec{x}$. This completes the induction.

Since $F_n = \mathcal{S}$ holds by definition, the claim follows. □

B.2 Orders with Auxiliary Variables

Next, we explain how two pseudo-Boolean formulas $\mathcal{O}_{\preceq}(\vec{u}, \vec{v}, \vec{a})$ and $\mathcal{S}_{\preceq}(\vec{u}, \vec{v}, \vec{a})$, together with the two disjoint lists of variables $\vec{z}$ and $\vec{a}$, define a preorder.

Definition 7. Let $\vec{u}$ and $\vec{v}$ be disjoint lists of variables of size n and let $\vec{a}$ be a list of auxiliary variables. Let $\mathcal{O}_{\preceq}(\vec{u}, \vec{v}, \vec{a})$ and $\mathcal{S}_{\preceq}(\vec{u}, \vec{v}, \vec{a})$ be two pseudo-Boolean formulas such that $\mathcal{S}_{\preceq}$ is a specification over $\vec{a}$.

Then we define the relation $\preceq$ over the domain of total assignments to a list of variables $\vec{z}$ of size n as follows: For assignments α, β we let $\alpha \preceq \beta$ hold, if and only if there exists an assignment ϱ to the variables $\vec{a}$, such that

$$\mathcal{S}_{\preceq}(\vec{z}\restriction_{\alpha}, \vec{z}\restriction_{\beta}, \vec{a}\restriction_{\varrho}) \wedge \mathcal{O}_{\preceq}(\vec{z}\restriction_{\alpha}, \vec{z}\restriction_{\beta}, \vec{a}\restriction_{\varrho})$$

evaluates to true.

To ensure that $\mathcal{O}_{\preceq}$ and $\mathcal{S}_{\preceq}$ actually define a preorder, we require cutting planes proofs that show reflexivity, i.e., $\emptyset \vdash \alpha \preceq \alpha$, and transitivity, i.e., $\alpha \preceq \beta \wedge \beta \preceq \gamma \vdash \alpha \preceq \gamma$. To write these proof obligations using the cutting planes proof system, which cannot handle an existentially quantified conclusion, we can use the specification as a premise. The specification premise essentially tells us which auxiliary variables the existential quantifier should pick. In particular, for reflexivity, the proof obligation is

$$\mathcal{S}_{\preceq}(\vec{x}, \vec{x}, \vec{a}) \vdash \mathcal{O}_{\preceq}(\vec{x}, \vec{x}, \vec{a})\,. \tag{21}$$

For transitivity, the proof obligation is

$$\begin{aligned} &\mathcal{S}_{\preceq}(\vec{x},\vec{y},\vec{a}) \cup O_{\preceq}(\vec{x},\vec{y},\vec{a}) \cup \mathcal{S}_{\preceq}(\vec{y},\vec{z},\vec{b}) \\ &\qquad \cup O_{\preceq}(\vec{y},\vec{z},\vec{b}) \cup \mathcal{S}_{\preceq}(\vec{x},\vec{z},\vec{c}) \vdash O_{\preceq}(\vec{x},\vec{z},\vec{c})\,. \end{aligned} \tag{22}$$

Intuitively, (22) says that if the circuits defining the auxiliary variables are correctly evaluated, which is encoded by the premises $\mathcal{S}_{\preceq}(\vec{x},\vec{y},\vec{a}) \cup \mathcal{S}_{\preceq}(\vec{y},\vec{z},\vec{b}) \cup \mathcal{S}_{\preceq}(\vec{x},\vec{z},\vec{c})$, then transitivity should hold, i.e., $O_{\preceq}(\vec{x},\vec{y},\vec{a}) \cup O_{\preceq}(\vec{y},\vec{z},\vec{b}) \vdash O_{\preceq}(\vec{x},\vec{z},\vec{c})$. However, if the auxiliary variables are not correctly set, then no claims are made.

These proof obligations ensure that $\preceq$ is a preorder:

Lemma 7. *If $O_{\preceq}$ and $\mathcal{S}_{\preceq}$ satisfy Equations* (21) *and* (22)*, then $\preceq$ as defined by $O_{\preceq}$ and $\mathcal{S}_{\preceq}$ is a preorder.*

Proof. We need to show that $\preceq$ is reflexive and transitive.

(Reflexivity.) Let α be a total assignment to the variables $\vec{z}$. We have to show that $\alpha \preceq \alpha$ holds, that is, there is an assignment ϱ to the variables $\vec{a}$ such that

$$\mathcal{S}_{\preceq}(\vec{z}{\upharpoonright_\alpha},\vec{z}{\upharpoonright_\alpha},\vec{a}{\upharpoonright_\varrho}) \wedge O_{\preceq}(\vec{z}{\upharpoonright_\alpha},\vec{z}{\upharpoonright_\alpha},\vec{a}{\upharpoonright_\varrho}) \tag{23}$$

is satisfied. From Lemma 1, it follows that we can choose an extension ϱ to α, such that $\mathcal{S}_{\preceq}(\vec{z}{\upharpoonright_\alpha},\vec{z}{\upharpoonright_\alpha},\vec{a}{\upharpoonright_\varrho})$ is satisfied. Since this suffices to satisfy the preconditions of Equation 21, it follows that also $O_{\preceq}(\vec{z}{\upharpoonright_\alpha},\vec{z}{\upharpoonright_\alpha},\vec{a}{\upharpoonright_\varrho})$ is satisfied, which suffices to conclude that $\alpha \preceq \alpha$ holds.

(Transitivity.) Let α,β,γ be total assignments to the variables $\vec{z}$. We have to show that if $\alpha \preceq \beta$ and $\beta \preceq \gamma$ hold, then $\alpha \preceq \gamma$ holds. That is we can assume that there are assignments ϱ_1,ϱ_2 to $\vec{a}$ such that

$$\begin{aligned} &\mathcal{S}_{\preceq}(\vec{z}{\upharpoonright_\alpha},\vec{z}{\upharpoonright_\beta},\vec{a}{\upharpoonright_{\varrho_1}}) \cup O_{\preceq}(\vec{z}{\upharpoonright_\alpha},\vec{z}{\upharpoonright_\beta},\vec{a}{\upharpoonright_{\varrho_1}}) \\ &\quad \cup \mathcal{S}_{\preceq}(\vec{z}{\upharpoonright_\beta},\vec{z}{\upharpoonright_\gamma},\vec{a}{\upharpoonright_{\varrho_2}}) \cup O_{\preceq}(\vec{z}{\upharpoonright_\beta},\vec{z}{\upharpoonright_\gamma},\vec{a}{\upharpoonright_{\varrho_2}}) \end{aligned} \tag{24}$$

is satisfied. From Lemma 1, it follows that we can choose an extension ϱ_3, such that $\mathcal{S}_{\preceq}(\vec{z}{\upharpoonright_\alpha},\vec{z}{\upharpoonright_\gamma},\vec{a}{\upharpoonright_{\varrho_3}})$ is satisfied. Since this suffices to satisfy the preconditions of Equation (22), it follows that also $O_{\preceq}(\vec{z}{\upharpoonright_\alpha},\vec{z}{\upharpoonright_\gamma},\vec{a}{\upharpoonright_{\varrho_3}})$ is satisfied, from which we conclude that $\alpha \preceq \gamma$ holds. □

B.3 Validity

We extend the configurations of the proof system to $(\mathcal{C},\mathcal{D},O_{\preceq},\mathcal{S}_{\preceq},\vec{z},\vec{a},v)$, where, as explained in [BGMN23], v denotes the current upper bound on the objective f. In particular, we extend the notion of *weak-(F,f)-validity* from [BGMN23] to our new configurations. For this, we first define the relation $\preceq_f$ as follows:

$$\alpha \preceq_f \beta \quad \text{iff} \quad \alpha \preceq \beta \;\wedge\; f{\upharpoonright_\alpha} \leq f{\upharpoonright_\beta}\ . \tag{25}$$

We define the strict order $\prec_f$ by defining $\alpha \prec_f \beta$ to mean that both $\alpha \preceq_f \beta$ and $\beta \npreceq_f \alpha$ hold.

Definition 8. A configuration $(\mathcal{C},\mathcal{D},O_{\preceq},\mathcal{S}_{\preceq},\vec{z},\vec{a},v)$ is *weakly-(F,f)-valid* if the following conditions hold:

1. For every $v' < v$, it holds that if $F \cup \{f \leq v'\}$ is satisfiable, then $\mathcal{C} \cup \{f \leq v'\}$ is satisfiable.

2. For every total assignment α satisfying the constraints $\mathcal{C} \cup \{f \leq v-1\}$, there exists a total assignment $\alpha' \preceq_f \alpha$ satisfying $\mathcal{C} \cup \mathcal{D} \cup \{f \leq v-1\}$.

Furthermore, we assume that for any configuration $(\mathcal{C}, \mathcal{D}, \mathcal{O}_\preceq, \mathcal{S}_\preceq, \vec{z}, \vec{a}, v)$ the following conditions holds:

1. $\mathcal{O}_\preceq$ and $\mathcal{S}_\preceq$ refer to formulas for which Equations (21) and (22) have been successfully proven.

2. $\mathcal{S}_\preceq$ is a specification over $\vec{a}$.

3. The variables $\vec{a}$ only occur in $\mathcal{O}_\preceq$ and $\mathcal{S}_\preceq$, and these variables are disjoint from $\vec{z}$.

These invariants only concern $\mathcal{O}_\preceq$, $\mathcal{S}_\preceq$, $\vec{a}$ and $\vec{z}$. Whenever the order is changed, it is ensured that these invariants hold, and other rules do not change these components of the configuration.

Observe that due to these invariants, satisfying assignments for $\mathcal{C} \cup \mathcal{D}$ do not need to assign the variables $\vec{a}$. In the following, we assume that such assignments are indeed defined only over the domain $\mathrm{var}(\mathcal{C} \cup \mathcal{D})$.

B.4 Dominance-Based Strengthening Rule

As in the original system, the dominance rule allows adding a constraint C to the derived set using witness ω if from the premises $\mathcal{C} \cup \mathcal{D} \cup \{\neg C\}$ we can derive $\mathcal{C}\restriction_\omega$ and show that $\alpha \circ \omega \prec \alpha$ holds for all assignments α satisfying $\mathcal{C} \cup \mathcal{D} \cup \{\neg C\}$. To show $\alpha \circ \omega \prec \alpha$, we separately show that $\alpha \circ \omega \preceq \alpha$ and $\alpha \not\preceq \alpha \circ \omega$. To show that $\alpha \circ \omega \preceq \alpha$, we have to show that $\mathcal{O}_\preceq(\vec{z}\restriction_\omega, \vec{z}, \vec{a})$, assuming that the circuit defining the auxiliary variables $\vec{a}$ has been evaluated correctly, which is encoded by the specification $\mathcal{S}_\preceq(\vec{z}\restriction_\omega, \vec{z}, \vec{a})$. This leads to the proof obligation

$$\begin{aligned} &\mathcal{C} \cup \mathcal{D} \cup \{\neg C\} \cup \{f \leq v-1\} \\ &\qquad \cup \mathcal{S}_\preceq(\vec{z}\restriction_\omega, \vec{z}, \vec{a}) \vdash \mathcal{O}_\preceq(\vec{z}\restriction_\omega, \vec{z}, \vec{a})\,. \end{aligned} \tag{26}$$

To show that $\alpha \not\preceq \alpha \circ \omega$, we have to show that $\neg\mathcal{O}_\preceq(\vec{z}, \vec{z}\restriction_\omega, \vec{a})$ assuming $\mathcal{S}_\preceq(\vec{z}, \vec{z}\restriction_\omega, \vec{a})$. However, since $\neg\mathcal{O}_\preceq(\vec{z}, \vec{z}\restriction_\omega, \vec{a})$ is not necessarily a pseudo-Boolean formula (due to the negation), we instead show that we can derive contradiction from $\mathcal{O}_\preceq(\vec{z}, \vec{z}\restriction_\omega, \vec{a})$, leading to the proof obligation:

$$\begin{aligned} &\mathcal{C} \cup \mathcal{D} \cup \{\neg C\} \cup \{f \leq v-1\} \\ &\qquad \cup \mathcal{S}_\preceq(\vec{z}, \vec{z}\restriction_\omega, \vec{a}) \cup \mathcal{O}_\preceq(\vec{z}, \vec{z}\restriction_\omega, \vec{a}) \vdash \bot\,. \end{aligned} \tag{27}$$

The following lemma shows that these proof obligations indeed imply $\alpha \circ \omega \preceq \alpha$ and $\alpha \not\preceq \alpha \circ \omega$, respectively:

Lemma 8. *Let G be a formula and ω a witness with* $\mathrm{supp}(\omega) \subseteq \mathrm{var}(G)$. *Furthermore, let $\vec{a} \cap \mathrm{var}(G) = \emptyset$ and $\mathcal{S}_{\preceq}$ be a specification over $\vec{a}$. Also, let $\mathcal{O}_{\preceq}$ and $\mathcal{S}_{\preceq}$ define a pre-order $\preceq$. Then the following hold:*

1. *If $G \cup \mathcal{S}_{\preceq}(\vec{z}\restriction_{\omega}, \vec{z}, \vec{a}) \vdash \{f\restriction_{\omega} \leq f\} \cup \mathcal{O}_{\preceq}(\vec{z}\restriction_{\omega}, \vec{z}, \vec{a})$, then for each assignment α satisfying G, $\alpha \circ \omega \preceq_f \alpha$ holds.*
2. *If $G \cup \mathcal{S}_{\preceq}(\vec{z}, \vec{z}\restriction_{\omega}, \vec{a}) \cup \mathcal{O}_{\preceq}(\vec{z}, \vec{z}\restriction_{\omega}, \vec{a}) \vdash \perp$ holds, then for each assignment α satisfying G, $\alpha \not\preceq_f \alpha \circ \omega$ holds.*

Proof. 1. Let α be an assignment satisfying G. By Lemma 1, we can extend α to ϱ such that $\mathcal{S}_{\preceq}(\vec{z}\restriction_{\alpha\circ\omega}, \vec{z}\restriction_{\alpha}, \vec{a}\restriction_{\varrho})$ is satisfied. This means that ϱ satisfies the preconditions of

$$G \cup \mathcal{S}_{\preceq}(\vec{z}\restriction_{\omega}, \vec{z}, \vec{a}) \vdash \{f\restriction_{\omega} \leq f\} \cup \mathcal{O}_{\preceq}(\vec{z}\restriction_{\omega}, \vec{z}, \vec{a})\,, \tag{28}$$

so also $\mathcal{O}_{\preceq}(\vec{z}\restriction_{\varrho\circ\omega}, \vec{z}\restriction_{\varrho}, \vec{a}\restriction_{\varrho})$ holds. Since α and ϱ coincide on $\vec{z}$ (which holds since $\vec{a}$ and $\vec{z}$ are disjoint), it follows that $\mathcal{O}_{\preceq}(\vec{z}\restriction_{\alpha\circ\omega}, \vec{z}\restriction_{\alpha}, \vec{a}\restriction_{\varrho})$ holds, which is sufficient to conclude that $\alpha \circ \omega \preceq \alpha$ holds. In addition, ϱ satisfies $\{f\restriction_{\omega} \leq f\}$. Since ϱ and α coincide on the variables of f, this implies that $f\restriction_{\alpha\circ\omega} \leq f_{\alpha}$, so together with $\alpha \circ \omega \preceq \alpha$ we conclude that $\alpha \circ \omega \preceq_f \alpha$ holds, as required.

2. Let α be an assignment satisfying G. We argue by contradiction. Suppose that $\alpha \preceq \alpha \circ \omega$ holds. Then there exists an assignment ϱ to $\vec{a}$ such that

$$\mathcal{S}_{\preceq}(\vec{z}_{\alpha}, \vec{z}\restriction_{\alpha\circ\omega}, \vec{a}\restriction_{\varrho}) \cup \mathcal{O}_{\preceq}(\vec{z}_{\alpha}, \vec{z}\restriction_{\alpha\circ\omega}, \vec{a}\restriction_{\varrho})\,.$$

Define α' to coincide with ϱ on $\vec{a}$, and with α otherwise. Since $\vec{a}$ and $\vec{z}$ are disjoint, α' then coincides with α on $\vec{z}$. Since $\vec{a} \cap \mathrm{var}(G) = \emptyset$, α' also coincides with α on $\mathrm{var}(G)$. Hence, α' satisfies G and

$$\mathcal{S}_{\preceq}(\vec{z}_{\alpha'}, \vec{z}\restriction_{\alpha'\circ\omega}, \vec{a}\restriction_{\alpha'}) \cup \mathcal{O}_{\preceq}(\vec{z}_{\alpha'}, \vec{z}\restriction_{\alpha'\circ\omega}, \vec{a}\restriction_{\alpha'})\,,$$

so α' satisfies the preconditions of $G \cup \mathcal{S}_{\preceq}(\vec{z}, \vec{z}\restriction_{\omega}, \vec{a}) \cup \mathcal{O}_{\preceq}(\vec{z}, \vec{z}\restriction_{\omega}, \vec{a}) \vdash \perp$, which is clearly contradictory. Hence, we have $\alpha \not\preceq \alpha \circ \omega$, which implies $\alpha \not\preceq_f \alpha \circ \omega$. □

Hence, we define the dominance rule as follows:

Definition 9 (Dominance-based strengthening with specification)**.** We can transition from the configuration $(\mathcal{C}, \mathcal{D}, \mathcal{O}_{\preceq}, \mathcal{S}_{\preceq}, \vec{z}, \vec{a}, v)$ to $(\mathcal{C}, \mathcal{D} \cup \{C\}, \mathcal{O}_{\preceq}, \mathcal{S}_{\preceq}, \vec{z}, \vec{a}, v)$ using the dominance rule if the following conditions are met:

1. The constraint C does not contain variables in $\vec{a}$.
2. There is a witness ω for which $\mathrm{image}(\omega) \cap \vec{a} = \emptyset$ holds.
3. We have cutting planes proofs for the following:

$$\begin{aligned} &\mathcal{C} \cup \mathcal{D} \cup \{\neg C\} \cup \{f \leq v-1\} \cup \mathcal{S}_{\preceq}(\vec{z}\restriction_{\omega}, \vec{z}, \vec{a}) \\ &\qquad \vdash C\restriction_{\omega} \cup \{f\restriction_{\omega} \leq f\} \cup \mathcal{O}_{\preceq}(\vec{z}\restriction_{\omega}, \vec{z}, \vec{a})\,, \end{aligned} \tag{29}$$

$$\begin{aligned} &\mathcal{C} \cup \mathcal{D} \cup \{\neg C\} \cup \{f \leq v-1\} \\ &\qquad \cup \mathcal{S}_{\preceq}(\vec{z}, \vec{z}\restriction_{\omega}, \vec{a}) \cup \mathcal{O}_{\preceq}(\vec{z}, \vec{z}\restriction_{\omega}, \vec{a}) \vdash \perp\,. \end{aligned} \tag{30}$$

Using Lemma 8, we can show that the dominance rule preserves the invariants required by weak validity:

Lemma 9. *If we can transition from the configuration $(\mathcal{C}, \mathcal{D}, \mathcal{O}_{\preceq}, \mathcal{S}_{\preceq}, \vec{z}, \vec{a}, v)$ to the configuration $(\mathcal{C}, \mathcal{D} \cup \{C\}, \mathcal{O}_{\preceq}, \mathcal{S}_{\preceq}, \vec{z}, \vec{a}, v)$ by the dominance rule, and the configuration $(\mathcal{C}, \mathcal{D}, \mathcal{O}_{\preceq}, \mathcal{S}_{\preceq}, \vec{z}, \vec{a}, v)$ is weakly-(F, f)-valid, then the configuration $(\mathcal{C}, \mathcal{D} \cup \{C\}, \mathcal{O}_{\preceq}, \mathcal{S}_{\preceq}, \vec{z}, \vec{a}, v)$ is also weakly-(F, f)-valid.*

Proof. Since $F, f, \mathcal{C}$, and v are not affected, item 1. in Definition 8 still holds. Hence, it remains to show that item 2. holds, i.e., that for every total assignment α that satisfies $\mathcal{C} \cup \{f \leq v - 1\}$ there exists a total assignment $\alpha' \preceq_f \alpha$ that satisfies $\mathcal{C} \cup \mathcal{D} \cup \{C\} \cup \{f \leq v - 1\}$.

Assume towards a contradiction that it does not hold. Let S denote the set of total assignments α that

1. satisfy $\mathcal{C} \cup \{f \leq v - 1\}$ and
2. admit no $\alpha' \preceq_f \alpha$ satisfying $\mathcal{C} \cup \mathcal{D} \cup \{C\} \cup \{f \leq v - 1\}$.

By our assumption, S is non-empty. Since $\prec_f$ is a strict order and S is finite and non-empty, S contains a $\prec_f$-minimal element. Let α be some $\prec_f$-minimal assignment in S. Since $(\mathcal{C}, \mathcal{D}, \mathcal{O}_{\preceq}, \mathcal{S}_{\preceq}, \vec{z}, \vec{a}, v)$ is weakly-(F, f)-valid, there exists some $\alpha_1 \preceq_f \alpha$ that satisfies $\mathcal{C} \cup \mathcal{D} \cup \{f \leq v - 1\}$. We know that α_1 cannot satisfy C since $\alpha \in S$. Hence, α_1 satisfies

$$G = \mathcal{C} \cup \mathcal{D} \cup \{\neg C\} \cup \{f \leq v - 1\}\,.$$

Since G does not contain variables of $\vec{a}$, both points of Lemma 8 apply by (29) and (30), respectively. Hence, it follows that $\alpha_1 \circ \omega \preceq_f \alpha_1$ and $\alpha_1 \npreceq_f \alpha_1 \circ \omega$ hold, which together imply that $\alpha_1 \circ \omega \prec_f \alpha_1$. Note that (29) also implies that α_1 satisfies $\mathcal{C}{\upharpoonright_\omega}$, so $\alpha_1 \circ \omega$ satisfies $\mathcal{C}$. Moreover, α_1 satisfies $f \leq v - 1$ and $f{\upharpoonright_\omega} \leq f$, which together imply $f{\upharpoonright_\omega} \leq v - 1$. Hence, $\alpha_1 \circ \omega$ satisfies $f \leq v - 1$.

Let $\alpha_2 = \alpha_1 \circ \omega$. Then α_2 satisfies $\mathcal{C} \cup \{f \leq v - 1\}$. Since $\alpha_2 \prec_f \alpha_1$ and $\alpha_1 \preceq_f \alpha$, we have $\alpha_2 \prec_f \alpha$. Since α is a $\prec_f$-minimal element of S, this implies that $\alpha_2 \notin S$. Hence, since α_2 does satisfy $\mathcal{C} \cup \{f \leq v - 1\}$, it follows that α_2 does admit an $\alpha' \preceq_f \alpha_2$ satisfying $\mathcal{C} \cup \mathcal{D} \cup \{C\} \cup \{f \leq v - 1\}$. But it also holds that $\alpha' \preceq_f \alpha_2 \prec_f \alpha$, so $\alpha' \preceq_f \alpha$. This contradicts $\alpha \in S$, finishing the proof. □

B.5 Redundance-Based Strengthening Rule

We also modify the redundance rule to work in our extended proof system. Similarly to the dominance rule, we can use $\mathcal{S}_{\preceq}(\vec{z}{\upharpoonright_\omega}, \vec{z}, \vec{a})$ as an extra premise in our proof obligations.

Definition 10 (Redundance-based strengthening with specification)**.** We can transition from $(\mathcal{C}, \mathcal{D}, \mathcal{O}_{\preceq}, \mathcal{S}_{\preceq}, \vec{z}, \vec{a}, v)$ to $(\mathcal{C}, \mathcal{D} \cup \{C\}, \mathcal{O}_{\preceq}, \mathcal{S}_{\preceq}, \vec{z}, \vec{a}, v)$ using the redundance rule if the following conditions are met:

1. The constraint C does not contain variables in $\vec{a}$.
2. There is a witness ω for which $\mathrm{image}(\omega) \cap \vec{a} = \emptyset$ holds.

3. We have cutting planes proof that the following holds:

$$\begin{aligned} &\mathcal{C} \cup \mathcal{D} \cup \{\neg C\} \cup \{f \leq v-1\} \cup \mathcal{S}_{\leq}(\vec{z}\restriction_{\omega}, \vec{z}, \vec{a}) \\ &\quad \vdash (\mathcal{C} \cup \mathcal{D} \cup \{C\})\restriction_{\omega} \cup \{f\restriction_{\omega} \leq f\} \cup \mathcal{O}_{\leq}(\vec{z}\restriction_{\omega}, \vec{z}, \vec{a}) . \end{aligned} \tag{31}$$

The redundance rule preserves weak validity:

Lemma 10. *If we can transition from the configuration $(\mathcal{C}, \mathcal{D}, \mathcal{O}_{\leq}, \mathcal{S}_{\leq}, \vec{z}, \vec{a}, v)$ to the configuration $(\mathcal{C}, \mathcal{D} \cup \{C\}, \mathcal{O}_{\leq}, \mathcal{S}_{\leq}, \vec{z}, \vec{a}, v)$ by the redundance rule, and the configuration $(\mathcal{C}, \mathcal{D}, \mathcal{O}_{\leq}, \mathcal{S}_{\leq}, \vec{z}, \vec{a}, v)$ is weakly-(F, f)-valid, then the configuration $(\mathcal{C}, \mathcal{D} \cup \{C\}, \mathcal{O}_{\leq}, \mathcal{S}_{\leq}, \vec{z}, \vec{a}, v)$ is also weakly-(F, f)-valid.*

Proof. As for the dominance rule, F, f, $\mathcal{C}$, and v are not affected, so item 1. in Definition 8 still holds. Hence, it remains to show that item 2. holds, i.e., that for every total assignment α that satisfies $\mathcal{C} \cup \{f \leq v-1\}$ there exists a total assignment $\alpha' \preceq_f \alpha$ that satisfies

$$\mathcal{C} \cup \mathcal{D} \cup \{C\} \cup \{f \leq v-1\} .$$

Let α be an assignment that satisfies $\mathcal{C} \cup \{f \leq v-1\}$. Since $(\mathcal{C}, \mathcal{D}, \mathcal{O}_{\leq}, \mathcal{S}_{\leq}, \vec{z}, \vec{a}, v)$ is weakly-(F, f)-valid, there exists some $\alpha_1 \preceq_f \alpha$ that satisfies $\mathcal{C} \cup \mathcal{D} \cup \{f \leq v-1\}$. If α_1 also satisfies C, then we are done. Otherwise, α_1 satisfies

$$G = \mathcal{C} \cup \mathcal{D} \cup \{\neg C\} \cup \{f \leq v-1\} .$$

Since G does not contain variables of $\vec{a}$, point 1 of Lemma 8 apply by (31). Hence, it follows that $\alpha_1 \circ \omega \preceq_f \alpha_1$. Equation (31) also implies that α_1 satisfies $(\mathcal{C} \cup \mathcal{D} \cup \{C\})\restriction_{\omega}$, so $\alpha_1 \circ \omega$ satisfies $\mathcal{C} \cup \mathcal{D} \cup \{C\}$. Finally, α_1 satisfies $f \leq v-1$ and $f\restriction_{\omega} \leq f$, which together imply $f\restriction_{\omega} \leq v-1$. Hence, $\alpha_1 \circ \omega$ satisfies $f \leq v-1$. Then $\alpha_1 \circ \omega \preceq_f \alpha_1 \preceq_f \alpha$ and $\alpha_1 \circ \omega$ satisfies $\mathcal{C} \cup \mathcal{D} \cup \{C\} \cup \{f \leq v-1\}$, so $\alpha_1 \circ \omega$ is an assignment showing that item 2. in Definition 8 holds for α. □

Appendix C Proof Logging in Satsuma

In this appendix, we show how our proof system using auxiliary variables is used to log the symmetry-breaking clauses generates by SATSUMA. This is done in two steps:

1. Defining the lexicographical order using auxiliary variables and proving its transitivity and reflexivity.

2. Deriving the symmetry-breaking clauses using the dominance rule and the loaded lexicographical order.

C.1 Defining the Lexicographical Order

We first discuss how to define the lexicographical order on sequences of length n using auxiliary variables. In Lemma 11, we provide an encoding of the lexicographical order using auxiliary variables, and show that this indeed encodes the

lexicographical order. In Lemma 12, we explain how to translate this encoding to pseudo-Boolean constraints. In Lemma 13, we then show that the reflexivity follows by reverse unit propagation (RUP), while Lemma 14 shows that transitivity can be shown using a cutting planes derivation of size $O(n)$. Finally, this implies Theorem 15, which states that we can define the lexicographical order in VeriPB using a derivation of size $O(n)$.

Throughout this section, $n \in \mathbb{N}^+$ denotes the length of the sequences over which we define lexicographical order, x_i and y_i are Boolean variables, and a_i and d_i are (auxiliary) Boolean variables.

Lemma 11. *Let $x_1, \dots, x_n$ and $y_1, \dots, y_n$ be given sequences of Boolean variables. Define the Boolean variables $a_1, \dots, a_{n-1}$ and $d_1, \dots, d_n$ inductively by*

$$a_1 \iff (x_1 \geq y_1)\,, \tag{32}$$

$$a_{i+1} \Leftrightarrow (a_i \;\wedge\; x_{i+1} \geq y_{i+1})\,, \tag{33}$$

$$d_1 \Leftrightarrow (y_1 \geq x_1)\,, \tag{34}$$

$$d_{i+1} \Leftrightarrow (d_i \;\wedge\; (\overline{a_i} \;\vee\; y_{i+1} \geq x_{i+1}))\,. \tag{35}$$

Then d_n holds if and only if $(x_1, \dots, x_n) \preceq_{\text{lex}} (y_1, \dots, y_n)$.

Proof. We show by induction on $i \geq 1$ that d_i holds if and only if $(x_1, \dots, x_i) \preceq_{\text{lex}} (y_1, \dots, y_i)$ (for $i \leq n$) and that $d_i \wedge a_i$ holds if and only if $(x_1, \dots, x_i) = (y_1, \dots, y_i)$ (for $i \leq n - 1$). The base case is immediate from the definitions of a_1 and d_1.

We proceed with the induction step. Note that we have to show two equivalences, so four implications.

d_{i+1} **implies $(\mathbf{x_1}, \dots, \mathbf{x_{i+1}}) \preceq_{\text{lex}} (\mathbf{y_1}, \dots, \mathbf{y_{i+1}})$.** Suppose that d_{i+1} holds. Then (35) implies that d_i and $\overline{a_i} \vee y_{i+1} \geq x_{i+1}$ hold. Since d_i holds, the induction hypothesis implies that

$$(x_1, \dots, x_i) \preceq_{\text{lex}} (y_1, \dots, y_i). \tag{36}$$

We split cases using $\overline{a_i} \vee y_{i+1} \geq x_{i+1}$:

- If $\overline{a_i}$, then $d_i \wedge a_i$ does not hold, so the induction hypothesis implies that $(x_1, \dots, x_i) \neq (y_1, \dots, y_i)$, which combined with (36) implies that $(x_1, \dots, x_{i+1}) \preceq_{\text{lex}} (y_1, \dots, y_{i+1})$.
- If $y_{i+1} \geq x_{i+1}$, then (36) also implies $(x_1, \dots, x_{i+1}) \preceq_{\text{lex}} (y_1, \dots, y_{i+1})$.

Hence, we have $(x_1, \dots, x_{i+1}) \preceq_{\text{lex}} (y_1, \dots, y_{i+1})$.

$(x_1, \dots, x_{i+1}) \preceq_{\text{lex}} (y_1, \dots, y_{i+1})$ **implies** d_{i+1}**.** Conversely, let us suppose that $(x_1, \dots, x_{i+1}) \preceq_{\text{lex}} (y_1, \dots, y_{i+1})$. Then also $(x_1, \dots, x_i) \preceq_{\text{lex}} (y_1, \dots, y_i)$, so the induction hypothesis implies that d_i holds. We consider two cases:

- If $(x_1, \dots, x_i) \neq (y_1, \dots, y_i)$, then $d_i \wedge a_i$ does not hold (by the induction hypothesis), so $\overline{a_i}$ holds (since d_i holds).
- If $(x_1, \dots, x_i) = (y_1, \dots, y_i)$, then $(x_1, \dots, x_{i+1}) \preceq_{\text{lex}} (y_1, \dots, y_{i+1})$ implies $y_{i+1} \geq x_{i+1}$.

Hence, $\overline{a_i} \vee y_{i+1} \geq x_{i+1}$ holds, which implies that d_{i+1} holds.

$d_{i+1} \wedge a_{i+1}$ **implies** $(x_1, \ldots, x_{i+1}) = (y_1, \ldots, y_{i+1})$**.** Suppose that $d_{i+1} \wedge a_{i+1}$ holds. Then also $d_i \wedge a_i$ holds (by (33) and (35)), so the induction hypothesis implies that $(x_1, \ldots, x_i) = (y_1, \ldots, y_i)$. It remains to show that $x_{i+1} = y_{i+1}$, which we show by showing both inequalities:

- a_{i+1} implies $x_{i+1} \geq y_{i+1}$ (by (33)).
- d_{i+1} implies $\overline{a_i} \vee y_{i+1} \geq x_{i+1}$ (by (35)), which using that a_i holds implies $y_{i+1} \geq x_{i+1}$.

Hence, we conclude that $(x_1, \ldots, x_{i+1}) = (y_1, \ldots, y_{i+1})$.

$(x_1, \ldots, x_{i+1}) = (y_1, \ldots, y_{i+1})$ **implies** $d_{i+1} \wedge a_{i+1}$**.** Conversely, let us suppose that $(x_1, \ldots, x_{i+1}) = (y_1, \ldots, y_{i+1})$. Then also $(x_1, \ldots, x_i) = (y_1, \ldots, y_i)$, so the induction hypothesis implies that $d_i \wedge a_i$ holds. Moreover, we have $x_{i+1} = y_{i+1}$. Then $y_{i+1} \geq x_{i+1}$ together with d_i implies d_{i+1} (by (35)), while $x_{i+1} \geq y_{i+1}$ together with a_i implies a_{i+1} (by (33)). We conclude that $d_{i+1} \wedge a_{i+1}$ holds.

This completes the induction and hence the proof. □

The constraints given in Lemma 11 are not written as pseudo-Boolean constraints, but they can be expressed using pseudo-Boolean constraints. For this, note that the constraint $r \Leftrightarrow C$, where r is a Boolean variable and $C \doteq \sum_i p_i x_i \geq P$ is a pseudo-Boolean constraint in normalized form, can be expressed using the pair of pseudo-Boolean constraints $P\overline{r} + \sum_i p_i x_i \geq P$ and $(\sum_i p_i - P + 1)r + \sum_i p_i \overline{x_i} \geq \sum_i p_i - P + 1$. We call a constraint of the form $r \Leftrightarrow C$ a *reification*, and the process of rewriting a reification to a pair of pseudo-Boolean constraints *expanding the reification*.

Lemma 12. *We can write the constraints* (32) *up to* (35) *as follows as pseudo-Boolean constraints:*

$$\overline{a_1} + x_1 + \overline{y_1} \geq 1\,, \tag{37}$$

$$2a_1 + \overline{x_1} + y_1 \geq 2\,, \tag{38}$$

$$3\overline{a_{i+1}} + 2a_i + x_{i+1} + \overline{y_{i+1}} \geq 3\,, \tag{39}$$

$$2a_{i+1} + 2\overline{a_i} + \overline{x_{i+1}} + y_{i+1} \geq 2\,, \tag{40}$$

$$\overline{d_1} + y_1 + \overline{x_1} \geq 1\,, \tag{41}$$

$$2d_1 + \overline{y_1} + x_1 \geq 2\,, \tag{42}$$

$$4\overline{d_{i+1}} + 3d_i + \overline{a_i} + y_{i+1} + \overline{x_{i+1}} \geq 4\,, \tag{43}$$

$$4d_{i+1} + 3\overline{d_i} + a_i + \overline{y_{i+1}} + x_{i+1} \geq 3\,. \tag{44}$$

Proof. For constraint (32), we rewrite $x_1 \geq y_1$ as $x_1 + \overline{y_1} \geq 1$, and then expand the reification.

For constraint (33), we note that $a_i \wedge x_{i+1} \geq y_{i+1}$ is equivalent to $a_i \wedge x_{i+1} + \overline{y_{i+1}} \geq 1$, which in turn is equivalent to $2a_i + x_{i+1} + \overline{y_{i+1}} \geq 3$. Then we expand the reification.

For constraint (34), we rewrite $y_1 \geq x_1$ as $y_1 + \overline{x_1} \geq 1$, and then expand the reification.

For constraint (35), we first note that $\overline{a_i} \lor y_{i+1} \geq x_{i+1}$ is equivalent to $\overline{a_i} \lor y_{i+1} + \overline{x_{i+1}} \geq 1$, which is in turn equivalent to $\overline{a_i} + y_{i+1} + \overline{x_{i+1}} \geq 1$. Hence, $d_i \land (\overline{a_i} \lor y_{i+1} \geq x_{i+1})$ is equivalent to the pseudo-Boolean constraint $3d_i + \overline{a_i} + y_{i+1} + \overline{x_{i+1}} \geq 4$. Finally, we expand the reification. □

We write $\mathcal{S}_{\preceq}(\vec{x}, \vec{y}, \vec{a}, \vec{d})$ for the constraints (37) up to (44) in Lemma 12. Note that $\mathcal{S}_{\preceq}$ has four arguments, rather than three as in Section 3.2, since we use two different symbols for the auxiliary variables to emphasize their different functions. We write $\mathcal{O}_{\preceq}(\vec{d})$ for the single constraint $d_n \geq 1$. Throughout the remainder of this appendix, we write $\preceq$ for the preorder with the specification $\mathcal{S}_{\preceq}(\vec{x}, \vec{y}, \vec{a}, \vec{d})$ and definition $\mathcal{O}_{\preceq}(\vec{d})$. Then Lemma 11 shows that the preorder $\preceq$ defined by $\mathcal{S}_{\preceq}$ and $\mathcal{O}_{\preceq}$ is indeed the lexicographical order.

Next, we show how we can prove reflexivity and transitivity of this order in VeriPB.

Lemma 13. *We can prove reflexivity of the order $\preceq$ using a single RUP step, requiring* $O(n)$ *propagations.*

Proof. Since proof goals are always shown by contradiction in VeriPB, we assume that $d_n \geq 1$ does not hold, i.e., that $\overline{d_n} \geq 1$ holds. For the reflexivity proof, we have access to the specification $\mathcal{S}_{\preceq}(\vec{x}, \vec{x}, \vec{a}, \vec{d})$. Under the substitution $y_i = x_i$, the constraints (42) and (44) simplify to

$$2d_1 \geq 1\,, \tag{45}$$

$$4d_{i+1} + 3\overline{d_i} + a_i \geq 2\,. \tag{46}$$

Then (45) propagates d_1 to 1, after which (46) inductively propagates d_{i+1} to 1 (for $1 \leq i \leq n-1$). After deriving d_n, we obtain a contradiction with $\overline{d_n} \geq 1$. □

Lemma 14. *We can prove transitivity of the order $\preceq$ using a cutting planes derivation of size* $O(n)$.

Proof. Let $\mathcal{S}_{\preceq}(\vec{x}, \vec{y}, \vec{a}, \vec{d})$, $\mathcal{S}_{\preceq}(\vec{y}, \vec{z}, \vec{b}, \vec{e})$ and $\mathcal{S}_{\preceq}(\vec{x}, \vec{z}, \vec{c}, \vec{f})$ be the specifications corresponding to the relations $(x_1, \ldots, x_n) \preceq (y_1, \ldots, y_n)$, $(y_1, \ldots, y_n) \preceq (z_1, \ldots, z_n)$, and $(x_1, \ldots, x_n) \preceq (z_1, \ldots, z_n)$, respectively. We also assume that $\mathcal{O}_{\preceq}(\vec{d})$ and $\mathcal{O}_{\preceq}(\vec{e})$ hold, which are the constraints $d_n \geq 1$ and $e_n \geq 1$. From this, we need to show that $\mathcal{O}_{\preceq}(\vec{f})$ holds, which is the constraint $f_n \geq 1$.

Overview. The proof consists of two main steps:

1. Deriving inductively that $d_i \geq 1$ for $1 \leq i \leq n$, and simplifying constraints (41) and (43) to account for the known values of the d_i-variables. Similarly, deriving that $e_i \geq 1$ for $1 \leq i \leq n$, and simplifying constraints using known values of the e_i-variables.

2. Deriving inductively that $f_i \geq 1$ for $1 \leq i \leq n$, for which we also inductively derive that c_i implies a_i and b_i.

> The intuition for why c_i implies a_i and b_i is as follows. We know that $(x_1, \dots, x_i) \preceq (y_1, \dots, y_i) \preceq (z_1, \dots, z_i)$, and c_i encodes that $(x_1, \dots, x_i) = (z_1, \dots, z_i)$. However, since $(y_1, \dots, y_i)$ is 'squeezed' in between $(x_1, \dots, x_i)$ and $(z_1, \dots, z_i)$, this equality means that also the equalities $(x_1, \dots, x_i) = (y_1, \dots, y_i)$ and $(y_1, \dots, y_i) = (z_1, \dots, z_i)$ hold, which is in turn encoded by a_i and b_i.

Since we use a constant number of cutting planes steps in the inductive step of these inductive derivations, the total number of cutting planes steps is $O(n)$. Since each constraint has bounded size, the size of the derivation is also $O(n)$.

Deriving $d_i \geq 1$ and simplifying constraints. We start by deriving that all d_i hold and simplify the constraints containing the d_i. Weakening constraint (43) on a_i, y_{i+1} and x_i yields $4\overline{d_{i+1}} + 3d_i \geq 1$. Hence, from $d_n \geq 1$ we can inductively prove $d_i \geq 1$ for $1 \leq i \leq n$. This in turn allows us to derive the constraints $\overline{a_i} + y_{i+1} + \overline{x_{i+1}} \geq 1$ (by adding $4 \cdot (d_{i+1} \geq 1)$ to (43) and weakening on d_i), and the constraint $y_1 + \overline{x_1} \geq 1$. Similarly, we derive $e_i \geq 1$, the constraints $\overline{b_i} + z_{i+1} + \overline{y_{i+1}} \geq 1$, and the constraint $z_1 + \overline{y_1} \geq 1$.

Deriving $f_i \geq 1$ and c_i implies a_i and b_i. We inductively show that $f_i \geq 1$, and that c_i implies a_i and b_i, i.e., that $a_i + \overline{c_i} \geq 1$ and $b_i + \overline{c_i} \geq 1$. For the base case, adding $y_1 + \overline{x_1} \geq 1$ and $z_1 + \overline{y_1} \geq 1$ yields $z_1 + \overline{x_1} \geq 1$, which in turn implies that $f_1 \geq 1$. Adding the three constraints

$$
\begin{aligned}
2a_1 + \overline{x_1} + y_1 &\geq 2\,, \\
\overline{c_1} + x_1 + \overline{z_1} &\geq 1\,, \\
z_1 + \overline{y_1} &\geq 1\,,
\end{aligned}
$$

and saturating yields $a_1 + \overline{c_1} \geq 1$. Similarly, we derive the implication $b_1 + \overline{c_1} \geq 1$, which completes the base case.

We proceed with the inductive step. We first derive that f_{i+1} holds. Adding the constraints

$$
\begin{aligned}
\overline{a_i} + y_{i+1} + \overline{x_{i+1}} &\geq 1\,, \\
\overline{b_i} + z_{i+1} + \overline{y_{i+1}} &\geq 1\,, \\
a_i + \overline{c_i} &\geq 1\,, \\
b_i + \overline{c_i} &\geq 1\,,
\end{aligned}
$$

and saturating, we get $\overline{c_i} + z_{i+1} + \overline{x_{i+1}} \geq 1$. Adding this constraint, $3 \cdot (f_i \geq 1)$ and $3f_{i+1} + 3\overline{f_i} + c_i + \overline{z_{i+1}} + x_{i+1} \geq 3$ and saturating then allows us to derive $f_{i+1} \geq 1$.

Finally, we have to show that c_{i+1} implies a_{i+1} and b_{i+1}. First note that $c_{i+1} \Rightarrow (c_i \wedge x_{i+1} \geq z_{i+1})$ implies the two constraints $c_{i+1} \Rightarrow c_i$ and $c_{i+1} \Rightarrow (x_{i+1} \geq z_{i+1})$. Indeed, weakening $3\overline{c_{i+1}} + 2c_i + x_{i+1} + \overline{z_{i+1}} \geq 3$ on x_{i+1} and z_{i+1} and saturating yields $\overline{c_{i+1}} + c_i \geq 1$, while weakening on c_i and saturating yields $\overline{c_{i+1}} + x_{i+1} + \overline{z_{i+1}} \geq 1$. Adding $\overline{c_{i+1}} + c_i \geq 1$ to $a_i + \overline{c_i} \geq 1$ and $b_i + \overline{c_i} \geq 1$ respectively yields $a_i + \overline{c_{i+1}} \geq 1$

and $b_i + \overline{c_{i+1}} \geq 1$. Adding the constraints

$$\begin{aligned}\overline{c_{i+1}} + x_{i+1} + \overline{z_{i+1}} &\geq 1\,,\\ 2a_{i+1} + 2\overline{a_i} + \overline{x_{i+1}} + y_{i+1} &\geq 2\,,\\ \overline{b_i} + z_{i+1} + \overline{y_{i+1}} &\geq 1\,,\end{aligned}$$

and saturating yields $\overline{c_{i+1}} + a_{i+1} + \overline{a_i} + \overline{b_i} \geq 1$. Then adding $a_i + \overline{c_{i+1}} \geq 1$ and $b_i + \overline{c_{i+1}} \geq 1$ and saturating yields $\overline{c_{i+1}} + a_{i+1} \geq 1$. We similarly derive $\overline{c_{i+1}} + b_{i+1} \geq 1$, which completes the inductive step and hence the proof. □

Together, this shows the following result:

Theorem 15. *We can define the lexicographical order over n variables in VeriPB using auxiliary variables and prove its transitivity and reflexivity using a VeriPB proof of size* $O(n)$ *that can be checked in time* $O(n)$.

C.2 Deriving the Symmetry-Breaking Clauses

Next, we show how to use our proof system in applications of the dominance rule where the witness is a symmetry σ of the input formula. Let $\mathrm{supp}(\sigma) = \{x_{i_1}, \ldots, x_{i_k}\}$ be the support of σ (where $i_1 \leq \ldots \leq i_k$). We want to derive symmetry breaking clauses encoding the lex-leader constraint

$$(x_{i_1}, \ldots, x_{i_k}) \preceq_{\mathrm{lex}} (\sigma(x_{i_1}), \ldots, \sigma(x_{i_k}))\,.$$

Throughout this appendix, we assume that the lexicographical order $\preceq$ over n variables is loaded. We first show how to define a circuit over variables s_l and t_l defining the lexicographical order over just $\mathrm{supp}(\sigma)$. Then we show how to do the actual symmetry breaking using the dominance rule. In particular, Lemma 16 and Lemma 17 show how we can prove the order proof goals corresponding to the dominance rule application. Lemma 18 then shows how to derive the symmetry breaking clauses. Together, this leads to Theorem 19, which shows that we can justify a symmetry σ of the input formula with $k = |\mathrm{supp}(\sigma)|$ using a VeriPB proof of size $O(k)$ that can be checked in time $O(n)$.

Defining a circuit. We first define at top level (using the redundance rule) a circuit similarly to the circuit defined in Lemma 11, using the following equations:

$$s_1 \Leftrightarrow (x_{i_1} \geq \sigma(x_{i_1}))\,, \tag{47}$$

$$s_{l+1} \Leftrightarrow (s_l \;\wedge\; x_{i_{l+1}} \geq \sigma(x_{i_{l+1}}))\,, \tag{48}$$

$$t_1 \Leftrightarrow (\sigma(x_{i_1}) \geq x_{i_1})\,, \tag{49}$$

$$t_{l+1} \Leftrightarrow (t_l \;\wedge\; (\overline{s_l} \vee \sigma(x_{i_{l+1}}) \geq x_{i_{l+1}}))\,, \tag{50}$$

where we have constraint (48) for $1 \leq l \leq k-2$ and constraint (50) for $1 \leq l \leq k-1$.

Intuitively, the variables s_l and t_l have the same role as the auxiliary variables a_{i_l} and d_{i_l} in the specification, respectively. As in the proof of Lemma 11, t_l hold if and only if $(x_{i_1}, \ldots, x_{i_l}) \preceq_{\mathrm{lex}} (\sigma(x_{i_1}), \ldots, \sigma(x_{i_l}))$, while $s_l \wedge t_l$ holds if and only if $(x_{i_1}, \ldots, x_{i_l}) = (\sigma(x_{i_1}), \ldots, \sigma(x_{i_l}))$.

As in Lemma 12, we can write these constraints as follows as pseudo-Boolean constraints:

$$\overline{s_1} + x_{i_1} + \overline{\sigma(x_{i_1})} \geq 1\,, \tag{51}$$
$$2s_1 + \overline{x_{i_1}} + \sigma(x_{i_1}) \geq 2\,, \tag{52}$$
$$3\overline{s_{l+1}} + 2s_l + x_{i_{l+1}} + \overline{\sigma(x_{i_{l+1}})} \geq 3\,, \tag{53}$$
$$2s_{l+1} + 2\overline{s_l} + \overline{x_{i_{l+1}}} + \sigma(x_{i_{l+1}}) \geq 2\,, \tag{54}$$
$$\overline{t_1} + \sigma(x_{i_1}) + \overline{x_{i_1}} \geq 1\,, \tag{55}$$
$$2t_1 + \overline{\sigma(x_{i_1})} + x_{i_1} \geq 2\,, \tag{56}$$
$$4\overline{t_{l+1}} + 3t_l + \overline{s_l} + \sigma(x_{i_{l+1}}) + \overline{x_{i_{l+1}}} \geq 4\,, \tag{57}$$
$$3t_{l+1} + 3\overline{t_l} + s_l + \overline{\sigma(x_{i_{l+1}})} + x_{i_{l+1}} \geq 3\,. \tag{58}$$

In total, this circuit consists of $4k - 2$ pseudo-Boolean constraints, and can be derived using the redundance rule in time O(1) per constraint. Note that for this the autoproving of the order proof goal of the redundance rule and the explicit loading of the specification discussed in Section 5 are essential to avoid a factor n overhead. Namely, these applications of the redundance rule witness over fresh variables only, so in particular not over the variables over which the order is loaded. Hence, the order proof goal follows directly from the reflexivity of the order, and these checker improvements allow us to detect this in time O(1), instead of wasting time O(n) on loading the specification and repeating the reflexivity proof.

Overview: the symmetry breaking clauses. We first state the symmetry breaking clauses:

$$s_1 + \overline{x_{i_1}} \geq 1\,, \tag{59}$$
$$s_{l+1} + \overline{s_l} + \overline{x_{i_{l+1}}} \geq 1\,, \tag{60}$$
$$s_1 + \sigma(x_{i_1}) \geq 1\,, \tag{61}$$
$$s_{l+1} + \overline{s_l} + \sigma(x_{i_{l+1}}) \geq 1\,, \tag{62}$$
$$\sigma(x_{i_1}) + \overline{x_{i_1}} \geq 1\,, \tag{63}$$
$$\overline{s_l} + \sigma(x_{i_{l+1}}) + \overline{x_{i_{l+1}}} \geq 1\,. \tag{64}$$

Note that a constraint whose coefficients and whose right hand side is 1 corresponds to a clause: for example, the constraint $s_1 + \overline{x_{i_1}} \geq 1$ is equivalent to the clause $s_1 \vee \overline{x_{i_1}}$.

In the symmetry breaking clauses, the variable s_l corresponds to the equality $(x_{i_1}, \ldots, x_{i_l}) = (\sigma(x_{i_1}), \ldots, \sigma(x_{i_l}))$. Clauses (59) up to (62) ensure that s_l propagates to true if this equality holds, while (63) and (64) provide the symmetry breaking constraint $\sigma(x_{i_1}) \geq x_{i_1}$ and that $\sigma(x_{i_{l+1}}) \geq x_{i_{l+1}}$ should hold if $(x_{i_1}, \ldots, x_{i_l}) = (\sigma(x_{i_1}), \ldots, \sigma(x_{i_l}))$.

Intuitively, to break a symmetry we will show using dominance that t_k and hence all t_l hold, and then we derive these clauses from the circuit.

Deriving the symmetry breaking clauses. Using the dominance rule, we now derive that $t_k \geq 1$. Let $C \doteq t_k \geq 1$. To apply the dominance rule, we have to show that

$$\mathcal{C} \cup \mathcal{D} \cup \{\neg C\} \cup \mathcal{S}_{\preceq}(\vec{x}\restriction_{\sigma}, \vec{x}, \vec{a}, \vec{d}) \vdash \mathcal{C}\restriction_{\sigma} \cup \mathcal{O}_{\preceq}(\vec{d}) \,, \tag{65}$$

$$\mathcal{C} \cup \mathcal{D} \cup \{\neg C\} \cup \mathcal{S}_{\preceq}(\vec{x}, \vec{x}\restriction_{\sigma}, \vec{a}, \vec{d}) \cup \mathcal{O}_{\preceq}(\vec{d}) \vdash \perp \,, \tag{66}$$

Since the witness σ is a symmetry of the formula, the proof goals corresponding to $\mathcal{C} \vdash \mathcal{C}\restriction_{\sigma}$ follow directly from $\mathcal{C}\restriction_{\sigma} = \mathcal{C}$. The next two lemmas explain how to prove the other two proof goals.

Note that within the subproof of the dominance rule, we can use the negation $\neg C \doteq \overline{t_k} \geq 1$, which intuitively means that $(x_{i_1}, \ldots, x_{i_k}) \not\preceq_{\text{lex}} (\sigma(x_{i_1}), \ldots, \sigma(x_{i_k}))$.

Lemma 16. *Assume that the lexicographical order $\preceq$ over n variables is loaded. Let σ be a symmetry of the input formula and let $k = |\operatorname{supp}(\sigma)|$. Then we can show the proof goal*

$$\mathcal{C} \cup \mathcal{D} \cup \{\neg C\} \cup \mathcal{S}_{\preceq}(\vec{x}\restriction_{\sigma}, \vec{x}, \vec{a}, \vec{d}) \vdash \mathcal{O}_{\preceq}(\vec{d}) \,,$$

using $\mathrm{O}(k)$ *RUP steps and cutting planes steps, where the RUP steps require* $\mathrm{O}(n)$ *propagations in total.*

Proof. Note that $\mathcal{O}_{\preceq}(\vec{d})$ is the single constraint $d_n \geq 1$. Since proof goals are always shown by contradiction, we assume that $\overline{d_n} \geq 1$ holds. This means that $(\sigma(x_{i_1}), \ldots, \sigma(x_{i_k})) \not\preceq_{\text{lex}} (x_{i_1}, \ldots, x_{i_k})$, but since $\preceq_{\text{lex}}$ is a complete order and we also have $(x_{i_1}, \ldots, x_{i_k}) \not\preceq_{\text{lex}} (\sigma(x_{i_1}), \ldots, \sigma(x_{i_k}))$ via the constraint $\neg C \doteq \overline{t_k} \geq 1$, we should be able to derive contradiction.

Overview. The proof consists of four main steps:

1. Rewriting the specification $\mathcal{S}_{\preceq}(x\restriction_{\sigma}, x, a, d)$ to only $\mathrm{O}(k)$ constraints involving auxiliary variables a_{i_l} and d_{i_l} and variables from $\operatorname{supp}(\sigma)$ only.

2. Showing that $s_l \Rightarrow d_{i_l}$ and $a_{i_l} \Rightarrow t_l$.

3. Showing that $t_{l+1} + \overline{t_l} + d_{i_l} \geq 1$ and $d_{i_{l+1}} + \overline{d_{i_l}} + t_l \geq 1$.

4. Showing $d_{i_l} + t_l \geq 1$ and $t_{l+1} + d_{i_l} \geq 1$ and $d_{i_{l+1}} + t_l \geq 1$.

The first of these steps requires $\mathrm{O}(k)$ RUP steps for which $\mathrm{O}(n)$ propagations are needed in total, and $\mathrm{O}(k)$ cutting planes steps. The remaining steps require $\mathrm{O}(k)$ RUP steps for which $\mathrm{O}(k)$ propagations are needed.

Rewriting the specification constraints. Write $S = \{i_1, \ldots, i_k\}$. There are two cases which are slightly different, depending on whether $1 \in S$ or not. Here we consider the case $1 \notin S$. Under the substitution σ, the specification constraints

$\mathcal{S}_{\leq}(x{\restriction}_{\sigma}, x, a, d)$ are (semantically) given by

$$a_1 \geq 1\,, \tag{67}$$
$$a_i \Leftrightarrow (a_{i-1} \;\wedge\; \sigma(x_i) \geq x_i)\,, \qquad (i \in S) \tag{68}$$
$$a_i \Leftrightarrow a_{i-1}\,, \qquad (i \notin S) \tag{69}$$
$$d_1 \geq 1\,, \tag{70}$$
$$d_i \Leftrightarrow (d_{i-1} \wedge (\overline{a_{i-1}} \vee x_i \geq \sigma(x_i)))\,, \qquad (i \in S) \tag{71}$$
$$d_i \Leftrightarrow d_{i-1}\,. \qquad (i \notin S) \tag{72}$$

Using the constraints (67) and (69), it follows by RUP that $a_{i_1 - 1} \geq 1$ and $a_{i_l - 1} \Leftrightarrow a_{i_{l-1}}$ for $2 \leq l \leq k$. Similarly, using the constraints (70) and (72), it follows by RUP that $d_{i_1-1} \geq 1$ and $d_{i_l-1} \Leftrightarrow d_{i_{l-1}}$ for $2 \leq l \leq k$.

Using these constraints, we can (using cutting planes steps) rewrite constraints (68) and (71) as

$$a_{i_1} \Leftrightarrow (\sigma(x_{i_1}) \geq x_{i_1})\,, \tag{73}$$
$$a_{i_{l+1}} \Leftrightarrow (a_{i_l} \;\wedge\; \sigma(x_{i_{l+1}}) \geq x_{i_{l+1}})\,, \tag{74}$$
$$d_{i_1} \Leftrightarrow (x_{i_1} \geq \sigma(x_{i_1}))\,, \tag{75}$$
$$d_{i_{l+1}} \Leftrightarrow (d_{i_l} \;\wedge\; (\overline{a_{i_l}} \;\vee\; x_{i_{l+1}} \geq \sigma(x_{i_{l+1}})))\,. \tag{76}$$

Showing that $s_l \Rightarrow d_{i_l}$ and $a_{i_l} \Rightarrow t_l$. We inductively derive using RUP that $s_l \Rightarrow d_{i_l}$, i.e., $\overline{s_l} + d_{i_l} \geq 1$. When showing this by RUP, the negation propagates s_l and $\overline{d_{i_l}}$.

For the base case, note that $\overline{d_{i_1}}$ propagates that the inequality $x_{i_1} \geq \sigma(x_{i_1})$ is false (i.e., that $\overline{x_{i_1}}$ and $\sigma(x_{i_1})$ hold), which yields together with s_1 a conflict in (51).

For the inductive step, note that s_{l+1} propagates s_l using (53), which by the induction hypothesis propagates d_{i_l}. Then $\overline{d_{i_{l+1}}}$ propagates that $\overline{a_{i_l}} \vee (x_{i_{l+1}} \geq \sigma(x_{i_{l+1}}))$ is false, which in particular propagates that $x_{i_{l+1}} \geq \sigma(x_{i_{l+1}})$ is false (i.e., that $\overline{x_{i_{l+1}}}$ and $\sigma(x_{i_{l+1}})$ hold), which yields together with s_{l+1} a conflict in (53).

In the same way, we derive using RUP that $a_{i_l} \Rightarrow t_l$.

Showing that $t_{l+1} + \overline{t_l} + d_{i_l} \geq 1$ and $d_{i_{l+1}} + \overline{d_{i_l}} + t_l \geq 1$. Next, we derive using RUP that $t_{l+1} + \overline{t_l} + d_{i_l} \geq 1$. Its negation propagates $\overline{t_{l+1}}$, t_l, and $\overline{d_{i_l}}$. This propagates s_l using (58). But then we have s_l and $\overline{d_{i_l}}$, which is a conflict in $s_l \Rightarrow d_{i_l}$. In the same way, we derive using RUP that $d_{i_{l+1}} + \overline{d_{i_l}} + t_l \geq 1$.

Showing $d_{i_l} + t_l \geq 1$ and $t_{l+1} + d_{i_l} \geq 1$ and $d_{i_{l+1}} + t_l \geq 1$. Next, we derive using RUP inductively that $d_{i_l} + t_l \geq 1$ and $t_{l+1} + d_{i_l} \geq 1$ and $d_{i_{l+1}} + t_l \geq 1$. The negated constraint propagates $\overline{d_{i_l}}$ and $\overline{t_l}$. For the base case, note that $\overline{d_{i_1}}$ propagates that the inequality $x_{i_1} \geq \sigma(x_{i_1})$ is false, which yields together with $\overline{t_1}$ a conflict in (56).

For the inductive step, we first show how to derive $t_{l+1} + d_{i_l} \geq 1$ and $d_{i_{l+1}} + t_l \geq 1$ using $d_{i_l} + t_l \geq 1$. The negation of $t_{l+1} + d_{i_l} \geq 1$ propagates $\overline{t_{l+1}}$ and $\overline{d_{i_l}}$, which propagates t_l using $t_l + d_{i_l} \geq 1$, which then yields a conflict in the constraint $t_{l+1} + \overline{t_l} + d_{i_l} \geq 1$.

The derivation of $d_{i_{l+1}} + t_l \geq 1$ is similar.

Next, we show how to derive $d_{i_{l+1}} + t_{l+1} \geq 1$. The negation of $d_{i_{l+1}} + t_{l+1} \geq 1$ propagates $\overline{t_{l+1}}$ and $\overline{d_{i_{l+1}}}$, which propagates t_l and d_{i_l} using $t_{l+1} + d_{i_l} \geq 1$ and $d_{i_{l+1}} + t_l \geq 1$. Then $\overline{d_{i_{l+1}}}$ and d_{i_l} propagate that $x_{i_{l+1}} \geq \sigma(x_{i_{l+1}})$ is false, but $\overline{t_{l+1}}$ and t_l propagate that $\sigma(x_{i_{l+1}}) \geq x_{i_{l+1}}$ is false, which is a conflict. Hence, we derive $d_{i_{l+1}} + t_{l+1} \geq 1$ by RUP.

Finally, using $d_{i_k} + t_k \geq 1$, constraints (72) which yield $d_{i_k} \Leftrightarrow d_n$, and the two constraints $\overline{d_n} \geq 1$ and $\overline{t_k} \geq 1$, we propagate to conflict, which completes the proof. □

Alternative derivation using cutting planes steps. For step 2, 3 and 4 in the proof of Lemma 16, we can also provide a derivation using cutting planes steps instead.

Showing that $s_l \Rightarrow d_{i_l}$ and $a_{i_l} \Rightarrow t_l$. We show this by induction. For the base case, adding constraint (51), i.e., $\overline{s_1} + x_{i_1} + \overline{\sigma(x_{i_1})} \geq 1$, and $2d_{i_1} + \overline{x_{i_1}} + \sigma(x_{i_1}) \geq 2$ (from (75)), and saturating yields $\overline{s_1} + d_{i_1} \geq 1$.

For the inductive step, we first add

$$3 \cdot \left(3\overline{s_{l+1}} + 2s_l + x_{i_{l+1}} + \overline{\sigma(x_{i_{l+1}})} \geq 3\right),$$
$$2 \cdot \left(3d_{i_{l+1}} + 3\overline{d_{i_l}} + a_{i_l} + \overline{\sigma(x_{i_{l+1}})} + x_{i_{l+1}} \geq 3\right),$$

and then weaken on a_{i_l}, $\sigma(x_{i_{l+1}})$, and $x_{i_{l+1}}$, which yields

$$9\overline{s_{l+1}} + 6s_l + 6d_{i_{l+1}} + 6\overline{d_{i_l}} \geq 7.$$

Then we add $6 \cdot (\overline{s_l} + d_{i_l} \geq 1)$ and saturate, which finally yields the constraint $\overline{s_{l+1}} + d_{i_{l+1}} \geq 1$

The derivation of $\overline{a_{i_l}} + t_l \geq 1$ is similar.

Showing that $t_{l+1} + \overline{t_l} + d_{i_l} \geq 1$ and $d_{i_{l+1}} + \overline{d_{i_l}} + t_l \geq 1$. To show the constraint $t_{l+1} + \overline{t_l} + d_{i_l} \geq 1$, we start from constraint (58), i.e.,

$$3t_{l+1} + 3\overline{t_l} + s_l + \overline{\sigma(x_{i_{l+1}})} + x_{i_{l+1}} \geq 3,$$

add $\overline{s_l} + d_{i_l} \geq 1$, weaken on $\sigma(x_{i_{l+1}})$ and $x_{i_{l+1}}$, and then saturate. The derivation of $d_{i_{l+1}} + \overline{d_{i_l}} + t_l \geq 1$ is similar.

Showing $d_{i_l} + t_l \geq 1$ and $t_{l+1} + d_{i_l} \geq 1$ and $d_{i_{l+1}} + t_l \geq 1$. We show this by induction. For the base case, adding constraint (56), i.e., $2t_{i_1} + x_{i_1} + \overline{\sigma(x_{i_1})} \geq 2$, and $2d_{i_1} + \overline{x_{i_1}} + \sigma(x_{i_1}) \geq 2$, and dividing by 2 yields $\overline{s_1} + d_{i_1} \geq 1$.

For the inductive step, note that adding $d_{i_l} + t_l \geq 1$ to $t_{l+1} + \overline{t_l} + d_{i_l} \geq 1$ and saturating yields $t_{l+1} + d_{i_l} \geq 1$.

Similarly, adding $d_{i_l} + t_l \geq 1$ to $d_{i_{l+1}} + \overline{d_{i_l}} + t_l \geq 1$ and saturating yields $d_{i_{l+1}} + t_l \geq 1$.

To derive $d_{i_{l+1}} + t_{l+1} \geq 1$, we start by adding

$$3t_{l+1} + 3\overline{t_l} + s_l + \overline{\sigma(x_{i_{l+1}})} + x_{i_{l+1}} \geq 3\,,$$
$$3d_{i_{l+1}} + 3\overline{d_{i_l}} + a_{i_l} + \sigma(x_{i_{l+1}}) + \overline{x_{i_{l+1}}} \geq 3\,,$$

and weakening on a_{i_l} and s_l, which yields

$$3t_{l+1} + 3\overline{t_l} + 3d_{i_{l+1}} + 3\overline{d_{i_l}} \geq 2\,.$$

Then adding $3 \cdot \left(t_{l+1} + d_{i_l} \geq 1\right)$ and $3 \cdot \left(d_{i_{l+1}} + t_l \geq 1\right)$ and saturating yields $d_{i_{l+1}} + t_{l+1} \geq 1$. □

We proceed by showing the second proof goal.

Lemma 17. *Assume that the lexicographical order $\preceq$ over n variables is loaded. Let σ be a symmetry of the input formula and let $k = |\operatorname{supp}(\sigma)|$. Then we can show the proof goal*

$$\mathcal{C} \cup \mathcal{D} \cup \{\neg C\} \cup \mathcal{S}_{\preceq}(\vec{x}, \vec{x}{\upharpoonright_\sigma}, \vec{a}, \vec{d}) \cup \mathcal{O}_{\preceq}(\vec{d}) \vdash \bot\,,$$

using $\mathrm{O}(k)$ *RUP steps and cutting planes steps, where the RUP steps require* $\mathrm{O}(n)$ *propagations in total.*

Proof. Note that $\neg C \doteq \overline{t_k} \geq 1$ still has the interpretation that $(x_{i_1}, \ldots, x_{i_k}) \not\preceq_{\text{lex}} (\sigma(x_{i_1}), \ldots, \sigma(x_{i_k}))$, while the premises $\mathcal{S}_{\preceq}(\vec{x}, \vec{x}{\upharpoonright_\sigma}, \vec{a}, \vec{d}) \cup \mathcal{O}_{\preceq}(\vec{d})$ have the interpretation that $(x_{i_1}, \ldots, x_{i_k}) \preceq_{\text{lex}} (\sigma(x_{i_1}), \ldots, \sigma(x_{i_k}))$, so we should be able to derive contradiction.

The idea of the proof is to show that a_{i_l} and s_l have the same interpretation and that d_{i_l} and t_l have the same interpretation. This then gives a contradiction since we have the premises $d_n \geq 1$ but $\overline{t_k} \geq 1$.

Overview. The proof consists of four main steps:

1. Rewriting the specification $\mathcal{S}_{\preceq}(x, x{\upharpoonright_\sigma}, a, d)$ to only $\mathrm{O}(k)$ constraints involving auxiliary variables a_{i_l} and d_{i_l} and variables from $\operatorname{supp}(\sigma)$ only.

2. Showing that $d_{i_l} \geq 1$.

3. Showing that $a_{i_l} + \overline{s_l} \geq 1$.

4. Showing that $t_l \geq 1$.

The first of these steps requires $\mathrm{O}(k)$ RUP steps for which $\mathrm{O}(n)$ propagations are needed in total, and $\mathrm{O}(k)$ cutting planes steps. The remaining steps require $\mathrm{O}(k)$ RUP steps for which $\mathrm{O}(k)$ propagations are needed.

Rewriting the specification constraints. Write $S = \{i_1, \ldots, i_k\}$. We again consider the case $1 \notin S$. Under the substitution σ, the specification constraints $\mathcal{S}_{\preceq}(x, x\upharpoonright_\sigma, a, d)$ are (semantically) given by

$$a_1 \geq 1\,, \tag{77}$$
$$a_i \Leftrightarrow (a_{i-1} \wedge x_i \geq \sigma(x_i))\,, \quad (i \in S) \tag{78}$$
$$a_i \Leftrightarrow a_{i-1}\,, \quad (i \notin S) \tag{79}$$
$$d_1 \geq 1\,, \tag{80}$$
$$d_i \Leftrightarrow (d_{i-1} \wedge (\overline{a_{i-1}} \vee \sigma(x_i) \geq x_i))\,, \quad (i \in S) \tag{81}$$
$$d_i \Leftrightarrow d_{i-1}\,. \quad (i \notin S) \tag{82}$$

In the same way as in Lemma 16, we can rewrite this to

$$a_{i_1} \Leftrightarrow (x_{i_1} \geq \sigma(x_{i_1}))\,, \tag{83}$$
$$a_{i_{l+1}} \Leftrightarrow (a_{i_l} \;\wedge\; x_{i_{l+1}} \geq \sigma(x_{i_{l+1}}))\,, \tag{84}$$
$$d_{i_1} \Leftrightarrow (\sigma(x_{i_1}) \geq x_{i_1})\,, \tag{85}$$
$$d_{i_{l+1}} \Leftrightarrow (d_{i_l} \;\wedge\; (\overline{a_{i_l}} \;\vee\; \sigma(x_{i_{l+1}}) \geq x_{i_{l+1}}))\,. \tag{86}$$

We also show by RUP using (82) that $d_{i_k} \Leftrightarrow d_n$.

When replacing a_{i_l} with s_l and d_{i_l} with t_l, constraints (83) up to (86) exactly match constraints (47) up to (50).

Showing that $d_{i_l} \geq 1$. Since we have the constraint $d_n \geq 1$, first $d_{i_k} \Leftrightarrow d_n$ propagates that $d_{i_k} \geq 1$, and then (86) inductively propagates $d_{i_l} \geq 1$ for all $1 \leq l \leq k$, starting with the high indices.

Showing that $a_{i_l} + \overline{s_l} \geq 1$. We show this inductively. The negation of $a_{i_l} + \overline{s_l} \geq 1$ propagates $\overline{a_{i_l}}$ and s_l.

For the base case, we note that $\overline{a_{i_1}}$ propagates that $x_{i_1} \geq \sigma(x_{i_1})$ is false (i.e., that $\overline{x_{i_1}}$ and $\sigma(x_{i_1})$ hold) using (83), which then yields together with s_1 a conflict in (47).

For the inductive step, we note that s_{l+1} propagates s_l using (48), which by the induction hypothesis propagates a_{i_l}. Together with $\overline{a_{i_{l+1}}}$ and (84) this then propagates the inequality $x_{i_{l+1}} \geq \sigma(x_{i_{l+1}})$ to false, which then yields together with s_{l+1} a conflict in (48).

Showing that $t_l \geq 1$. We show this inductively. For the base case, the negation $\overline{t_1} \geq 1$ propagates the inequality $\sigma(x_{i_1}) \geq x_{i_1}$ to false, which the propagates d_{i_1} to false, which is a contradiction.

For the inductive step, the negation $\overline{t_{l+1}} \geq 1$ propagates together with $t_l \geq 1$ using (50) that $\overline{s_l} \vee (\sigma(x_{i_{l+1}}) \geq x_{i_{l+1}})$ is false, i.e., that s_l, $\overline{\sigma(x_{i_{l+1}})}$ and $x_{i_{l+1}}$. Then s_l propagates a_{i_l}, which together means that $\overline{a_{i_l}} \vee (\sigma(x_{i_{l+1}}) \geq x_{i_{l+1}})$ is false. This is a conflict in (86), since we know that $d_{i_{l+1}}$ holds.

We finish the proof by noting that this shows that $t_k \geq 1$, which directly conflicts with the premise $\overline{t_k} \geq 1$. □

Alternative derivation using cutting planes steps. For step 3 and 4 in the proof of Lemma 17, we can also provide a derivation using cutting planes steps instead.

Showing that $a_{i_l} + \overline{s_l} \geq 1$**.** For the base case, we add the constraint $2a_{i_1} + \overline{x_{i_1}} + \sigma(x_{i_1}) \geq 2$ (from (83)) and the constraint $\overline{s_1} + x_{i_1} + \overline{\sigma(x_{i_1})} \geq 1$ (from (47)) and saturate.

For the inductive step, we add the constraints

$$2a_{i_{l+1}} + 2\overline{a_{i_l}} + \overline{x_{i_{l+1}}} + \sigma(x_{i_{l+1}}) \geq 2$$
$$3\overline{s_{l+1}} + 2s_l + x_{i_{l+1}} + \overline{\sigma(x_{i_{l+1}})} \geq 3$$
$$2 \cdot \left(a_{i_l} + \overline{s_l} \geq 1\right)$$

from (84), (48) and the induction hypothesis respectively, and then saturate, which yields $a_{i_{l+1}} + \overline{s_{l+1}} \geq 1$.

Showing that $t_l \geq 1$**.** For the base case, we add the constraint $2t_1 + \overline{\sigma(x_{i_1})} + x_{i_1} \geq 2$ (from (49)) and the constraint $\overline{d_{i_1}} + \overline{x_{i_1}} + \sigma(x_{i_1}) \geq 1$ (from (85)) and the constraint $d_{i_1} \geq 1$ and saturate, which yields $t_1 \geq 1$.

For the inductive step, we add the constraints

$$4\overline{d_{i_{l+1}}} + 3d_{i_l} + \overline{a_{i_l}} + \sigma(x_{i_{l+1}}) + \overline{x_{i_{l+1}}} \geq 4$$
$$3t_{l+1} + 3\overline{t_l} + s_l + \overline{\sigma(x_{i_{l+1}})} + x_{i_{l+1}} \geq 3$$
$$a_{i_l} + \overline{s_l} \geq 1$$

from (50), (86), and earlier derivations. This yields

$$4\overline{d_{i_{l+1}}} + 3d_{i_l} + 3t_{l+1} + 3\overline{t_l} \geq 4$$

Then we add the constraints $4 \cdot \left(d_{i_{l+1}} \geq 1\right)$ and $3 \cdot \left(t_l \geq 1\right)$ (from the induction hypothesis) and weaken away d_{i_l} and saturate, which yields $t_{l+1} \geq 1$. □

Finally, we show how to derive the symmetry breaking clauses (59) up to (64) from the constraint $t_k \geq 1$ that we derived using dominance.

Lemma 18. *Let σ be a symmetry of the input formula and let $k = |\operatorname{supp}(\sigma)|$. Then we can derive the symmetry breaking clauses* (59) *up to* (64) *from the constraint $t_k \geq 1$ using* O(k) *RUP steps and cutting planes steps.*

Proof. We can derive constraint (59), i.e., $s_1 + \overline{x_{i_1}} \geq 1$, by weakening constraint (52), i.e., $2s_1 + \overline{x_{i_1}} + \sigma(x_{i_1}) \geq 2$, on $\sigma(x_{i_1})$ and saturating. Similarly, we can derive (61), i.e., $s_1 + \sigma(x_{i_1}) \geq 1$, by weakening (52) on x_{i_1} and saturating.

For constraint (60), i.e., $s_{l+1} + \overline{s_l} + \overline{x_{i_{l+1}}} \geq 1$, we weaken (54), i.e., $2s_{l+1} + 2\overline{s_l} + \overline{x_{i_{l+1}}} + \sigma(x_{i_{l+1}}) \geq 2$, on $\sigma(x_{i_{l+1}})$ and saturate. Similarly, for (62), i.e., $s_{l+1} + \overline{s_l} + \sigma(x_{i_{l+1}}) \geq 1$, we weaken (54) on $x_{i_{l+1}}$ and saturate.

It remains to prove constraints (63), i.e., $\sigma(x_{i_1}) + \overline{x_{i_1}} \geq 1$, and (64), i.e., $\overline{s_l} + \sigma(x_{i_{l+1}}) + \overline{x_{i_{l+1}}} \geq 1$. For this, we first derive using RUP from $t_k \geq 1$ and (57) that $t_l \geq 1$ for $1 \leq l \leq k$ (starting with the high indices). Then adding $t_1 \geq 1$ to (55) yields (63), while adding $4 \cdot \left(t_{l+1} \geq 1\right)$ to (57), weakening on t_l and saturating yields (64). □

Together this shows

Theorem 19. *Assume that the lexicographical order $\preceq$ over n variables is loaded. Let σ be a symmetry of the input formula and let $k = |\mathrm{supp}(\sigma)|$. Then we can derive the symmetry breaking clauses* (59) *up to* (64) *using a proof of size* $\mathrm{O}(k)$ *that can be checked in time* $\mathrm{O}(n)$.

Appendix D Example Of Symmetry Breaking Proof

We present an example of a proof in the VeriPB format using auxiliary preorder variables. In order to keep the size of the proof somewhat manageable, we consider a toy example: the pigeonhole principle with 3 pigeons and 2 holes. This formula contains variables x_i, where x_1 and x_2 encode that pigeon 1 flies into hole 1 and 2, x_3 and x_4 encode that pigeon 2 flies into hole 1 and 2, and x_5 and x_6 encode that pigeon 3 flies into hole 1 and 2, respectively. This unsatisfiable formula claims that, first, each pigeon is in some hole:

$$x_1 \vee x_2 \qquad x_3 \vee x_4 \qquad x_5 \vee x_6\,,$$

and second, that each hole contains at most one pigeon:

$$\begin{array}{ccc} \overline{x_1} \vee \overline{x_3} & \overline{x_1} \vee \overline{x_5} & \overline{x_3} \vee \overline{x_5} \\ \overline{x_2} \vee \overline{x_4} & \overline{x_2} \vee \overline{x_6} & \overline{x_4} \vee \overline{x_6}\,. \end{array}$$

Our proof example breaks two symmetries of this formula:

$$\begin{aligned} \sigma &= \{x_1 \mapsto x_3,\ x_2 \mapsto x_4,\ x_3 \mapsto x_1,\ x_4 \mapsto x_2\}\,, \\ \tau &= \{x_1 \mapsto x_6,\ x_2 \mapsto x_5,\ x_3 \mapsto x_2, \\ &\qquad x_4 \mapsto x_1,\ x_5 \mapsto x_4,\ x_6 \mapsto x_3\}\,, \end{aligned}$$

where σ swaps pigeons 1 and 2, and τ moves every pigeon to the other hole, and renames it to the previous pigeon.

The proof starts by defining a preorder `lex6`, corresponding to the lexicographic order over a list of 6 variables.

```
pseudo-Boolean proof version 3.0
def_order lex6
vars
left u1 u2 u3 u4 u5 u6;
right v1 v2 v3 v4 v5 v6;
aux $a1 $a2 $a3 $a4 $a5 $d1 $d2 $d3 $d4 $d5 $d6;
end vars;
```

Here, variables u_i and v_i are declared as left-hand and right-hand variables for the preorder `lex6`. Furthermore, auxiliary preorder variables a_i and d_i are declared. Next, the specification is declared, following Lemma 12.

```
spec
red +1 ~$a1 +1 u1 +1 ~v1 >= 1 : $a1 -> 0;
red +2 $a1 +1 ~u1 +1 v1 >= 2 : $a1 -> 1;
red +3 ~$a2 +2 $a1 +1 u2 +1 ~v2 >= 3 : $a2 -> 0;
red +2 $a2 +2 ~$a1 +1 ~u2 +1 v2 >= 2 : $a2 -> 1;
```

Here, the constraint $\overline{a_1} + u_1 + \overline{v_1} \geq 1$ is first introduced in the specification by redundance with witness $\{a_1 \mapsto 0\}$. Since a_1 is a fresh variable at this point, this is correct. Next, the constraint $a_1 + \overline{u_1} + v_1 \geq 2$ is introduced by redundance with witness $\{a_1 \mapsto 1\}$. Note that this witness satisfies the previous constraint, so this is correct too. The specification then introduces constraints $3\overline{a_2} + 2a_1 + u_2 + \overline{v_2} \geq 3$ and $2a_2 + \overline{a_1} + \overline{u_2} + v_2 \geq 2$ with witnesses $\{a_2 \mapsto 0\}$ and $\{a_2 \mapsto 1\}$, which are correct for similar reasons. The rest of the specification is introduced in a similar way:

```
red +3 ~$a3 +2 $a2 +1 u3 +1 ~v3 >= 3 : $a3 -> 0;
red +2 $a3 +2 ~$a2 +1 ~u3 +1 v3 >= 2 : $a3 -> 1;
red +3 ~$a4 +2 $a3 +1 u4 +1 ~v4 >= 3 : $a4 -> 0;
red +2 $a4 +2 ~$a3 +1 ~u4 +1 v4 >= 2 : $a4 -> 1;
red +3 ~$a5 +2 $a4 +1 u5 +1 ~v5 >= 3 : $a5 -> 0;
red +2 $a5 +2 ~$a4 +1 ~u5 +1 v5 >= 2 : $a5 -> 1;
red +1 ~$d1 +1 ~u1 +1 v1 >= 1 : $d1 -> 0;
red +2 $d1 +1 u1 +1 ~v1 >= 2 : $d1 -> 1;
red +4 ~$d2 +3 $d1 +1 ~$a1 +1 ~u2 +1 v2 >= 4 : $d2 -> 0;
red +3 $d2 +3 ~$d1 +1 $a1 +1 u2 +1 ~v2 >= 3 : $d2 -> 1;
red +4 ~$d3 +3 $d2 +1 ~$a2 +1 ~u3 +1 v3 >= 4 : $d3 -> 0;
red +3 $d3 +3 ~$d2 +1 $a2 +1 u3 +1 ~v3 >= 3 : $d3 -> 1;
red +4 ~$d4 +3 $d3 +1 ~$a3 +1 ~u4 +1 v4 >= 4 : $d4 -> 0;
red +3 $d4 +3 ~$d3 +1 $a3 +1 u4 +1 ~v4 >= 3 : $d4 -> 1;
red +4 ~$d5 +3 $d4 +1 ~$a4 +1 ~u5 +1 v5 >= 4 : $d5 -> 0;
red +3 $d5 +3 ~$d4 +1 $a4 +1 u5 +1 ~v5 >= 3 : $d5 -> 1;
red +4 ~$d6 +3 $d5 +1 ~$a5 +1 ~u6 +1 v6 >= 4 : $d6 -> 0;
red +3 $d6 +3 ~$d5 +1 $a5 +1 u6 +1 ~v6 >= 3 : $d6 -> 1;
end spec;
```

Finally, the preorder constraints for `lex6` are introduced. As per Section C.1, these are a single constraint $d_6 \geq 1$.

```
def
+1 $d6 >= 1;
end def;
```

This section is followed by a transitivity proof. The proof starts by declaring some new variables.

```
transitivity
vars
fresh_right w1 w2 w3 w4 w5 w6 ;
fresh_aux_1 $b1 $b2 $b3 $b4 $b5 $e1 $e2 $e3 $e4 $e5 $e6;
fresh_aux_2 $c1 $c2 $c3 $c4 $c5 $f1 $f2 $f3 $f4 $f5 $f6;
end vars;
```

These are needed to account for a new set of preorder variables $\vec{w}$, and two new sets of preorder auxiliary variables $\vec{b}, \vec{e}$ and $\vec{c}, \vec{f}$. These are given by the lines `fresh_right`, `fresh_aux_1` and `fresh_aux_2`, respectively. This proof will show the following implication:

$$\mathcal{S}_{\preceq}(\vec{u},\vec{v},\vec{a},\vec{d}) \cup \mathcal{O}_{\preceq}(\vec{u},\vec{v},\vec{a},\vec{d}) \cup \mathcal{S}_{\preceq}(\vec{v},\vec{w},\vec{b},\vec{e}) \cup \mathcal{O}_{\preceq}(\vec{v},\vec{w},\vec{b},\vec{e}) \cup \mathcal{S}_{\preceq}(\vec{u},\vec{w},\vec{c},\vec{f}) \vdash \mathcal{O}_{\preceq}(\vec{u},\vec{w},\vec{c},\vec{f})$$

Copies of the specification and preorder constraints are implicitly brought into scope and assigned numerical identifiers. In this particular case, the specification contains 22 constraints, and the preorder contains a single constraint. Then, a total

of 68 constraints are implicitly introduced in the proof scope, and given identifiers in the same order as they were defined in the `spec` and `ord` sections:

- $\mathcal{S}_{\preceq}(\vec{u}, \vec{v}, \vec{a}, \vec{d})$ is mapped into identifers `1` through `22`.
- $\mathcal{S}_{\preceq}(\vec{v}, \vec{w}, \vec{b}, \vec{e})$ is mapped into identifiers `23` through `44`.
- $\mathcal{S}_{\preceq}(\vec{u}, \vec{w}, \vec{c}, \vec{f})$ is mapped into identifiers `45` through `66`.
- The single constraints in $\mathcal{O}_{\preceq}(\vec{u}, \vec{v}, \vec{a}, \vec{d})$ and $\mathcal{O}_{\preceq}(\vec{v}, \vec{w}, \vec{b}, \vec{e})$ are mapped into identifiers `67` and `68`, respectively.

```
proof
proofgoal #1
```

Next, the `proof` section shows the implication above. Each `proofgoal` line provides a proof for one constraint in $\mathcal{O}_{\preceq}(\vec{u}, \vec{w}, \vec{c}, \vec{f})$, in order of definition in the `def` section. In our case, we only must prove $f_6 \geq 1$, so only one `proofgoal` appears in the proof. Within this context, the negation of the goal, $\overline{f_6} \geq 1$, is automatically assigned the next available identifier, in this case `69`; to satisfy the proofgoal we must reach a contradiction.

```
pol 67 4 * 21 +;
rup +1 $d5 >= 1 : -1;
pol -2 $d5 w;
pol -2 4 * 19 +;
rup +1 $d4 >= 1 : -1;
pol -2 $d4 w;
```

The proof roughly follows Lemma 14. First, constraints

$$3d_i + \overline{a_i} + u_{i+1} + \overline{v_{i+1}} \geq 4 \tag{87}$$

$$d_i \geq 1 \tag{88}$$

$$\overline{a_i} + \overline{u_{i+1}} + v_{i+1} \geq 1 \tag{89}$$

are derived for $i = 5, \ldots, 1$. We can see this in the displayed fragment for $i = 5$. Constraint (87) is derived through the line `pol 67 4 * 21 +`. Lines starting with `pol` derive the result of a cutting planes proof by providing explicit operations in reverse Polish notation. This line first fetches constraint 67 (which is $d_6 \geq 1$), then scales it by 4, then adds the result with constraint 21 (i.e. $4\overline{d_6} + 3d_5 + \overline{a_5} + u_6 + \overline{v_6} \geq 4$). Constraint (88) is derived through the line `rup +1 $d5 >= 1 : -1`. Lines starting with `rup` apply a form of cutting planes proof search called *reverse unit propagation* (RUP) that can automatically identify some inferences. After the colon, a list of constraint identifiers can be provided to use as premises for proof search. Negative identifiers refer to identifiers relative to the current constraint, so -1 in this case refers to the constraint we just derived, namely (87). For premises F and a conclusion C, RUP tries to identify a contradiction in $F \cup \{\overline{C}\}$ by constraint propagation. In this case, this formula contains the constraints

$$3d_5 + \overline{a_5} + u_6 + \overline{v_6} \geq 4 \qquad\qquad \overline{d_5} \geq 1$$

which constraint propagation proves inconsistent. Finally, constraint (89) is derived through the line `pol -2 $d5 w`, which is reverse Polish notation for weakening constraint (87) on variable d_5.

```
pol -2 4 * 17 +;
rup +1 $d3 >= 1 : -1;
pol -2 $d3 w;
pol -2 4 * 15 +;
rup +1 $d2 >= 1 : -1;
pol -2 $d2 w;
pol -2 4 * 13 +;
rup +1 $d1 >= 1 : -1;
pol -2 $d1 w;
pol -2 11 +;
```

This process iterates on i until the following constraints are derived

$$\begin{aligned} d_5 &\geq 1 \quad (71), & \overline{a_5} + \overline{u_6} + v_6 &\geq 1 \quad (72), \\ d_4 &\geq 1 \quad (74), & \overline{a_4} + \overline{u_5} + v_5 &\geq 1 \quad (75), \\ d_3 &\geq 1 \quad (77), & \overline{a_3} + \overline{u_4} + v_4 &\geq 1 \quad (78), \\ d_2 &\geq 1 \quad (80), & \overline{a_2} + \overline{u_3} + v_3 &\geq 1 \quad (81), \\ d_1 &\geq 1 \quad (83), & \overline{a_1} + \overline{u_2} + v_2 &\geq 1 \quad (84), \\ \overline{u_1} + v_1 &\geq 1 \quad (85), \end{aligned}$$

where the last constraint is derived through the line `pol -2 11 +`. The proof then repeats the process above for variables b_i, e_i.

```
pol 68 4 * 43 +;
rup +1 $e5 >= 1 : -1;
pol -2 $e5 w;
pol -2 4 * 41 +;
rup +1 $e4 >= 1 : -1;
pol -2 $e4 w;
pol -2 4 * 39 +;
rup +1 $e3 >= 1 : -1;
pol -2 $e3 w;
pol -2 4 * 37 +;
rup +1 $e2 >= 1 : -1;
pol -2 $e2 w;
pol -2 4 * 35 +;
rup +1 $e1 >= 1 : -1;
pol -2 $e1 w;
pol -2 33 +;
```

This yields constraints

$$\begin{aligned} e_5 &\geq 1 \quad (87), & \overline{b_5} + \overline{v_6} + w_6 &\geq 1 \quad (88), \\ e_4 &\geq 1 \quad (90), & \overline{b_4} + \overline{v_5} + w_5 &\geq 1 \quad (91), \\ e_3 &\geq 1 \quad (93), & \overline{b_3} + \overline{v_4} + w_4 &\geq 1 \quad (94), \\ e_2 &\geq 1 \quad (96), & \overline{b_2} + \overline{v_3} + w_3 &\geq 1 \quad (97), \\ e_1 &\geq 1 \quad (99), & \overline{b_1} + \overline{v_2} + w_2 &\geq 1 \quad (100), \\ \overline{v_1} + w_1 &\geq 1 \quad (101). \end{aligned}$$

Next, the proof derives constraints A_i and B_i given by $a_i + \overline{c_i} \geq 1$ and $b_i + \overline{c_i} \geq 1$ for $i = 1, \ldots, 6$, respectively. This follows the inductive process detailed in Lemma 14.

```
pol 24 45 + 85 + s;
pol 47 u2 w w2 w s;
```

In the base case, the line `pol 2 45 + -1 + s` derives $a_1 + \overline{c_1} \geq 1$ by adding and saturating constraints

$$\overline{u_1} + v_1 + 2a_1 \geq 2$$
$$u_1 + \overline{w_1} + \overline{c_1} \geq 1$$
$$\overline{v_1} + w_1 \geq 1$$

where the two constraints above come from the specifications, and the one below is 101. Similarly, the line `pol 24 45 + 85 + s` derives $b_1 + \overline{c_1} \geq 1$.

```
pol -1 -3 +;
pol -2 -3 +;
pol 47 $c1 w s;
pol -2 100 + -1 + -3 2 * + 4 + s;
pol -4 84 + -2 + -3 2 * + 26 + s;
```

The induction case needs to derive some intermediate lemmas at each step.

- First, the constraint C_i given by $c_i + c_{i+1} \geq 1$ is obtained by weakening the specification constraint $u_{i+1} + \overline{w_{i+1}} + 2c_i + 3\overline{c_{i+1}} \geq 3$ on variables u_{i+1} and w_{i+1} and then saturating; in the fragment displayed, this is done for $i = 1$ by the line `pol 47 u2 w w2 w s`.
- Then, the constraints A'_i and B'_i given by $a_i + \overline{c_{i+1}} \geq 1$ and $b_i + \overline{c_{i+1}} \geq 1$ are derived by adding C_i with A_i and with B_i, respectively. In the displayed fragment, this is done for $i = 1$ by the lines `pol -1 -3 +` and `pol -2 -3 +`.
- The constraint L_i given by $u_{i+1} + \overline{w_{i+1}} + \overline{c_{i+1}} \geq 1$ is derived by weakening the specification constraint $u_{i+1} + \overline{w_{i+1}} + 2c_i + 3\overline{c_{i+1}} \geq 3$ on variable c_i and then saturating. In the displayed fragment, this is done for $i = 1$ by the line `pol 47 $c1 w s`.
- Finally, A_{i+1} is derived by adding the constraints $2 \cdot A'_i + B'_i + L_i$ with $\overline{b_i} + \overline{v_{i+1}} + w_{i+1} \geq 1$ (which was derived in the previous proof fragment) and $\overline{u_{i+1}} + \overline{v_{i+1}} + 2\overline{a_i} + 2a_{i+1} \geq 2$ (which is a specification constraint), and then saturating. The constraint B_{i+1} is analogously derived as well. In the displayed fragment, this is done for $i = 1$ by the lines `pol -2 100 + -1 + -3 2 * + 4 + s` and `pol -4 84 + -2 + -3 2 * + 26 + s`.

```
pol 49 u3 w w3 w s;
pol -1 -3 +;
pol -2 -3 +;
pol 49 $c2 w s;
pol -2 97 + -1 + -3 2 * + 6 + s;
pol -4 81 + -2 + -3 2 * + 28 + s;
pol 51 u4 w w4 w s;
pol -1 -3 +;
pol -2 -3 +;
pol 51 $c3 w s;
pol -2 94 + -1 + -3 2 * + 8 + s;
pol -4 78 + -2 + -3 2 * + 30 + s;
pol 53 u5 w w5 w s;
pol -1 -3 +;
pol -2 -3 +;
```

```
pol 53 $c4 w s;
pol -2 91 + -1 + -3 2 * + 10 + s;
pol -4 75 + -2 + -3 2 * + 32 + s;
pol 85 101 +;
```

Last, the proof derives the constraint F_i given by $f_i \geq 1$ for $i = 1, \ldots, 6$.

```
pol -1 56 + s;
pol 84 100 + 102 + 103 + s -1 3 * +;
pol -1 58 + s;
pol 81 97 + 108 + 109 + s -1 3 * +;
pol -1 60 + s;
pol 78 94 + 114 + 115 + s -1 3 * +;
pol -1 62 + s;
pol 75 91 + 120 + 121 + s -1 3 * +;
pol -1 64 + s;
pol 72 88 + 126 + 127 + s -1 3 * +;
pol -1 66 + s;
```

For the base case, can derive F_1 by adding constraints $\overline{u_1} + v_1 \geq 1$ and $\overline{v_1} + w_1 \geq 1$, which we derived as 85 and 101; this is the line `pol 85 101 +`. In the induction case, we first derive a lemma F_i' given by $\overline{u_{i+1}} + w_{i+1} + \overline{c_i} + 3f_i \geq 4$. This is obtained by first and then saturating the constraints

$$\begin{aligned}
\overline{a_i} + \overline{u_{i+1}} + v_{i+1} &\geq 1\,,\\
\overline{b_i} + \overline{u_{i+1}} + v_{i+1} &\geq 1\,,\\
a_i + \overline{c_i} &\geq 1\,,\\
b_i + \overline{c_i} &\geq 1\,,
\end{aligned}$$

resulting in $\overline{c_i} + \overline{u_{i+1}} + \overline{w_{i+1}}$; the two constraints above come from the first transitivity proof fragment; the two constraints below are A_i and B_i. The result is added to $3 \cdot F_i$, which yields F_i'. This is done for $i = 1$ by line `pol 84 100 + 102 + 103 + s -1 3 * +`. The constraint F_{i+1} can then be derived by adding F_i' to the specification constraint $u_{i+1} + \overline{w_{i+1}} + c_i + 3\overline{f_i} + 3\overline{f_{i+1}} \geq 3$ and saturating; this is done for $i = 1$ by the line `pol -1 58 + s`.

```
pol -1 69 +;
qed #1 : -1;
qed proof;
end transitivity;
```

Once F_6 has been derived, the line `pol -1 69 +` adds it to the negated proof goal $\overline{f_6} \geq 1$, creating a contradiction $0 \geq 1$. The line `qed #1 : -1` communicates this contradiction to the proof checker, finishing the transitivity proof. The reflexivity proof follows.

```
reflexivity
proof
proofgoal #1
rup >= 1;
qed #1 : -1;
qed proof;
end reflexivity;
```

The reflexivity section automatically identifies the constraints in $\mathcal{S}_{\leq}(\vec{u}, \vec{u}, \vec{a}, \vec{d})$ as constraints 1 through 22. We must provide a proof of $O_{\leq}(\vec{u}, \vec{u}, \vec{a}, \vec{d})$, which contains

a single constraint $d_6 \geq 1$, identified by the proofgoal #1. Similar to the transitivity proof, entering the `proofgoal #1` section negates this goal as $\overline{d_6} \geq 1$ and adds it to the premises as constraint 23. As shown by Lemma 13, this proof derives a contradiction in a single RUP step in line `rup >= 1`.

```
end def_order;
load_order lex6 x5 x6 x1 x2 x3 x4;
```

After this, the preorder `lex6` has been correctly defined, and the refutation of the pigeonhole principle starts, assigning its constraints identifiers 1 through 9. The line `load_order lex6 x5 x6 x1 x2 x3 x4` applies an order change rule with the preorder `lex6`. The list of variables there tells the proof checker to map formula variables to preorder variables as:

$$x_5 \mapsto u_1, v_1 \qquad x_6 \mapsto u_2, v_2 \qquad x_1 \mapsto u_3, v_3$$
$$x_2 \mapsto u_4, v_4 \qquad x_3 \mapsto u_5, v_5 \qquad x_3 \mapsto u_6, v_6$$

The proof immediately proceeds to break the symmetry σ. First, the symmetry breaker circuit is introduced.

```
red +1 ~s1 +1 x1 +1 ~x3 >= 1 : s1 -> 0;
red +2 s1 +1 ~x1 +1 x3 >= 2 : s1 -> 1;
red +3 ~s2 +2 s1 +1 x2 +1 ~x4 >= 3 : s2 -> 0;
red +2 s2 +2 ~s1 +1 ~x2 +1 x4 >= 2 : s2 -> 1;
red +3 ~s3 +2 s2 +1 x3 +1 ~x1 >= 3 : s3 -> 0;
red +2 s3 +2 ~s2 +1 ~x3 +1 x1 >= 2 : s3 -> 1;
red +1 ~t1 +1 ~x1 +1 x3 >= 1 : t1 -> 0;
red +2 t1 +1 x1 +1 ~x3 >= 2 : t1 -> 1;
red +4 ~t2 +3 t1 +1 ~s1 +1 ~x2 +1 x4 >= 4 : t2 -> 0;
red +3 t2 +3 ~t1 +1 s1 +1 x2 +1 ~x4 >= 3 : t2 -> 1;
red +4 ~t3 +3 t2 +1 ~s2 +1 ~x3 +1 x1 >= 4 : t3 -> 0;
red +3 t3 +3 ~t2 +1 s2 +1 x3 +1 ~x1 >= 3 : t3 -> 1;
red +4 ~t4 +3 t3 +1 ~s3 +1 ~x4 +1 x2 >= 4 : t4 -> 0;
red +3 t4 +3 ~t3 +1 s3 +1 x4 +1 ~x2 >= 3 : t4 -> 1;
```

These correspond to constraints (51) through (58). Each constraint is introduced by redundance-based strengthening. First, the constraint $C_1 = \overline{s_1} + x_1 + \overline{x_3} \geq 1$ is derived with witness $\omega_1 = \{s_1 \mapsto 0\}$; this corresponds to constraint (51). The check $\mathcal{C} \cup \mathcal{D} \cup \{\overline{C_1}\} \vdash (\mathcal{C} \cup \mathcal{D} \cup \{C_1\})\restriction_{\omega_1}$ succeeds trivially, since s_1 does not occur in $\mathcal{C} \cup \mathcal{D}$ and $C_1\restriction_{\omega_1}$ is a tautology. For the same reason, the check $\mathcal{S}_{\preceq}(\vec{x}\restriction_{\omega_1}, \vec{x}, \vec{a}, \vec{d}) \vdash \mathcal{O}_{\preceq}(\vec{x}\restriction_{\omega_1}, \vec{x}, \vec{a}, \vec{d})$ can be skipped, since this is the same as the reflexivity check.

Next, the constraint $C_2 = 2s_1 + \overline{x_1} + x_3 \geq 2$ is derived with witness $\omega_2 = \{s_1 \mapsto 1\}$; this corresponds to constraint (52). The check $\mathcal{C} \cup \mathcal{D} \cup \{\overline{C_2}\} \vdash (\mathcal{C} \cup \mathcal{D} \cup \{C_2\})\restriction_{\omega_2}$ succeeds trivially. The reasons are the same as above for all constraints except the recently introduced C_1; in that case simply observe that C_1 is the result of weakening $\overline{C_2}$ (i.e. $2\overline{s_1} + x_1 + \overline{x_3} \geq 3$) on variable x_3.

A similar reasoning proves all remaining redundance-based strengthening steps. They receive constraint identifiers 10 through 23. Once these circuit constraints have been introduced, the proof is ready to introduce the symmetry breaker by dominance-based strengthening.

```
dom +1 t4  >= 1 : x1 -> x3 x2 -> x4 x3 -> x1 x4 -> x2  : subproof
```

This line starts the dominance proof to derive the constraint C given by $t_4 \geq 1$ with witness σ. Note that all the constraints derived above by redundance-based strengthening have been derived into $\mathcal{D}$; this becomes relevant for the checks elicited by dominance-based strengthening:

$$\mathcal{C} \cup \mathcal{D} \cup \{\neg C\} \cup \mathcal{S}_{\preceq}(\vec{x}\restriction_\sigma, \vec{x}, \vec{a}, \vec{d}) \vdash \mathcal{C}\restriction_\sigma \cup \mathcal{O}_{\preceq}(\vec{x}\restriction_\sigma, \vec{x}, \vec{a}, \vec{d})$$

$$\mathcal{C} \cup \mathcal{D} \cup \{\neg C\} \cup \mathcal{S}_{\preceq}(\vec{x}, \vec{x}\restriction_\sigma, \vec{a}, \vec{d}) \cup \mathcal{O}_{\preceq}(\vec{x}, \vec{x}\restriction_\sigma, \vec{a}, \vec{d}) \vdash \bot$$

The single constraint in $\mathcal{O}_{\preceq}(\vec{z}\restriction_\sigma, \vec{z}, \vec{a})$ in the first check is assigned proof goal identifier `#1`, and the $\bot$ in the second check is assigned proof goal identifier `#2`. The goals $\mathcal{C}\restriction_\sigma$ in the first check are assigned as proof goal identifiers their constraint identifiers from $\mathcal{C}$, without a "#"; we actually do not need to prove these, since σ is a symmetry of $\mathcal{C}$, so $\mathcal{C}\restriction_\sigma = \mathcal{C}$ holds. The `subproof` context above introduces the constraint $\neg C$ given by $\overline{t_4} \geq 1$ with identifier `24`.

```
scope leq
proofgoal #1
```

The scope `leq` introduces the constraints $\mathcal{S}_{\preceq}(\vec{x}\restriction_\sigma, \vec{x}, \vec{a}, \vec{d})$ from the first check with constraint identifiers `25` through `46`: and `proofgoal #1` selects the only constraint in $\mathcal{O}_{\preceq}(\vec{x}\restriction_\sigma, \vec{x}, \vec{a}, \vec{d})$, namely $d_6 \geq 1$, as a proofgoal; its negation is introduced with constraint identifier `47`. First the proof tries to derive the simplified specification constraints:

$$\begin{aligned} \overline{a_3} + x_3 + \overline{x_1} &\geq 1 \\ 2a_3 + \overline{x_3} + x_1 &\geq 2 \\ 3\overline{a_4} + 2a_3 + x_4 + \overline{x_2} &\geq 3 \\ 2a_4 + 2\overline{a_3} + \overline{x_4} + x_2 &\geq 2 \\ 3\overline{a_5} + 2a_4 + x_1 + \overline{x_3} &\geq 3 \\ 2a_5 + 2\overline{a_4} + \overline{x_1} + x_3 &\geq 2 \end{aligned} \qquad \begin{aligned} \overline{d_3} + x_1 + \overline{x_3} &\geq 1 \\ 2d_3 + \overline{x_1} + x_3 &\geq 2 \\ 4\overline{d_4} + 3d_3 + \overline{a_3} + x_2 + \overline{x_4} &\geq 4 \\ 3d_4 + 3\overline{d_3} + a_3 + \overline{x_2} + x_4 &\geq 3 \\ 4\overline{d_5} + 3d_4 + \overline{a_4} + x_3 + \overline{x_1} &\geq 4 \\ 3d_5 + 3\overline{d_4} + a_4 + \overline{x_3} + x_1 &\geq 3 \\ 4\overline{d_6} + 3d_5 + \overline{a_5} + x_4 + \overline{x_2} &\geq 4 \\ 3d_6 + 3\overline{d_5} + a_5 + \overline{x_4} + x_2 &\geq 3\,. \end{aligned}$$

First the reification $a_3 \iff x_3 \geq x_1$ is obtained.

```
rup +1 ~$d6 >= 1;
rup +1 $a2 >= 1;
pol 29 ~$a2 2 * + s;
pol 30 -2 2 * +;
```

Since $\sigma(x_5) = x_5$ and $\sigma(x_6) = x_6$, $\mathcal{S}_{\preceq}(\vec{x}\restriction_\sigma, \vec{x}, \vec{a}, \vec{d})$ contains the constraints $a_1 \geq 1$ and $2a_2 + 2\overline{a_i} \geq 1$, from which we can derive the lemma $a_2 \geq 1$ by RUP. Then from the literal axiom $\overline{a_2} \geq 2$ and specification constraint $3\overline{a_3} + 2a_2 + x_3 + \overline{x_1} \geq 3$ we can derive $\overline{a_3} + x_3 + \overline{x_1} \geq 1$. Similarly, from the lemma $a_1 \geq 1$ and the specification constraint $2a_3 + 2\overline{a_2} + \overline{x_3} + x_1 \geq 2$ we derive $2a_3 + \overline{x_3} + x_1 \geq 2$.

Next, we obtain the reification $a_4 \iff a_3 \wedge (x_4 \geq x_2)$.

```
rup +1 ~$a3 +1 $a3 >= 1;
rup +1 $a3 +1 ~$a3 >= 1;
pol 31 -2 2 * +;
pol 32 -2 2 * +;
```

The proof first tries to derive $a_i \iff a_j$ where i is the index for a right after the previous support variable, and j is the index for a at the next support variable. Since in this case $i = j = 3$, the proof actually degenerates into deriving two tautologies by RUP. Next, the proof derives constraints $3\overline{a_4} + 2a_3 + x_4 + \overline{x_2} \geq 3$ and $2a_4 + 2\overline{a_3} + \overline{x_4} + x_2 \geq 2$ by operating over some specification constraints and the reification $a_i \iff a_j$. These constraints actually already appear in the specification $\mathcal{S}_{\preceq}(\vec{x}\restriction_\sigma, \vec{x}, \vec{a}, \vec{d})$, so we do not really derive anything new. The proof then repeats this last step for the variable a_5.

```
rup +1 ~$a4 +1 $a4 >= 1;
rup +1 $a4 +1 ~$a4 >= 1;
pol 33 -2 2 * +;
pol 34 -2 2 * +;
rup +1 ~$a5 +1 $a5 >= 1;
rup +1 $a5 +1 ~$a5 >= 1;
```

A very similar approach derives the simplified reifications of the d_i variables.

```
rup +1 $d2 >= 1;
pol 39 ~$d2 3 * + 49 + s;
pol 40 -2 3 * + ~$a2 +;
rup +1 ~$d3 +1 $d3 >= 1;
rup +1 $d3 +1 ~$d3 >= 1;
pol 41 -2 3 * + -14 +;
pol 42 -2 3 * + -16 +;
rup +1 ~$d4 +1 $d4 >= 1;
rup +1 $d4 +1 ~$d4 >= 1;
pol 43 -2 3 * + -14 +;
pol 44 -2 3 * + -16 +;
rup +1 ~$d5 +1 $d5 >= 1;
rup +1 $d5 +1 ~$d5 >= 1;
pol 45 -2 3 * + -14 +;
pol 46 -2 3 * + -16 +;
```

The proof now proves the following constraints

$$d_3 + \overline{s_1} \geq 1 \qquad t_1 + \overline{a_3} \geq 1$$
$$d_4 + \overline{s_2} \geq 1 \qquad t_2 + \overline{a_4} \geq 1$$
$$d_5 + \overline{s_3} \geq 1 \qquad t_3 + \overline{a_5} \geq 1$$

by RUP.

```
rup +1 $d3 +1 ~s1 >= 1;
rup +1 $d4 +1 ~s2 >= 1;
rup +1 $d5 +1 ~s3 >= 1;
rup +1 t1 +1 ~$a3 >= 1;
rup +1 t2 +1 ~$a4 >= 1;
rup +1 t3 +1 ~$a5 >= 1;
```

Here, checking that $d_3 + \overline{s_1} \geq 1$ is implied by RUP can be done by assuming literals $\overline{d_3}$ and s_1 are true, and then propagating. The simplified specification constraint $2d_3 + \overline{x_1} + x_3 \geq 2$ then propagates $\overline{x_1}$ and x_3. This in turn yields a conflict with the symmetry breaker circuit constraint $\overline{s_1} + x_1 + \overline{x_3} \geq 1$. To show that $d_4 + \overline{s_2} \geq 2$ is implied by RUP, we similarly assume literals $\overline{d_4}$ and s_2 are true. Then, the symmetry breaker circuit constraint $3\overline{s_2} + 2s_1 + x_2 + \overline{x_4} \geq 3$ propagates s_1, and so the previously derived constraint propagates d_3. Finally, the simplified specification constraint $3d_4 + 3\overline{d_3} + a_3 + x_2 + \overline{x_4} \geq 3$ propagates $\overline{x_2}$ and x_4, which in turn conflict

with the same symmetry breaker circuit constraint above. Similar RUP steps derive the rest of the constraints above, as well as the rest of the proof.

```
rup +1 $d3 +1 ~s1 >= 1;
rup +1 $d4 +1 ~s2 >= 1;
rup +1 $d5 +1 ~s3 >= 1;
rup +1 t1 +1 ~$a3 >= 1;
rup +1 t2 +1 ~$a4 >= 1;
rup +1 t3 +1 ~$a5 >= 1;
rup +1 t2 +1 ~t1 +1 $d3 >= 1;
rup +1 t3 +1 ~t2 +1 $d4 >= 1;
rup +1 t4 +1 ~t3 +1 $d5 >= 1;
rup +1 $d4 +1 ~$d3 +1 t1 >= 1;
rup +1 $d5 +1 ~$d4 +1 t2 >= 1;
rup +1 $d6 +1 ~$d5 +1 t3 >= 1;
rup +1 $d3 +1 t1 >= 1;
rup +1 $d3 +1 t2 >= 1;
rup +1 $d4 +1 t1 >= 1;
rup +1 $d4 +1 t2 >= 1;
rup +1 $d4 +1 t3 >= 1;
rup +1 $d5 +1 t2 >= 1;
rup +1 $d5 +1 t3 >= 1;
rup +1 $d5 +1 t4 >= 1;
rup +1 $d6 +1 t3 >= 1;
rup +1 $d6 +1 t4 >= 1;
rup >= 1;
qed #1 : -1;
end scope;
```

Finally, the proof derives $d_6 + t_4 \geq 1$, which immediately yields a contradiction: $\overline{t_4} \geq 1$ is assumed as the negation of the current dominance constraint, and $\overline{d_6} \geq 1$ is assumed as the negation of the current proof goal.

```
rup >= 1;
qed #1 : -1;
end scope;
```

Next, the proof shows the second dominance check.

```
scope geq
proofgoal #2
```

The scope `geq` reintroduces the constraints $\mathcal{S}_{\preceq}(\vec{x}\upharpoonright_\sigma, \vec{x}, \vec{a}, \vec{d})$ with constraint identifiers `100` through `121`, just like `leq`. Additionally, the single constraint in $\mathcal{O}_{\preceq}(\vec{x}\upharpoonright_\sigma, \vec{x}, \vec{a}, \vec{d})$ is introduced as well with identifier `122`. The proof itself proceeds similarly to the `leq` check, by deriving constraints that connect $\vec{a}$, $\vec{d}$, $\vec{s}$ and $\vec{t}$. The RUP steps follow closely Lemma 17.

```
rup +1 $d5 >= 1;
rup +1 $d4 >= 1;
rup +1 $d3 >= 1;
rup +1 ~s1 +1 $a3 >= 1;
rup +1 ~s2 +1 $a4 >= 1;
rup +1 ~s3 +1 $a5 >= 1;
rup +1 t1 >= 1;
rup +1 t2 >= 1;
rup +1 t3 >= 1;
```

The proof eventually derives $t_3 \geq 1$ and $s_3 + \overline{a_5} \geq 1$. This leads to a contradiction, since the assumptions $\overline{t_4} \geq 1$ and $d_6 \geq 1$ then propagate x_4, $\overline{x_2}$ and s_3 from the symmetry breaker circuit constraint $3t_4 + 3\overline{t_3} + s_3 + x_4 + \overline{x_2} \geq 3$. The constraint

$s_3 + \overline{a_5} \geq 1$ then propagates a_5, and then the constraint $4\overline{d_6} + 3d_5 + \overline{a_5} + x_2 + \overline{x_4} \geq 4$ yields a conflict. This concludes the dominance proof.

```
rup >= 1;
qed #2 : -1;
end scope;
qed dom;
```

The proof next simplifies the constraints and deletes the circuit, leaving only some constraints that use the $\vec{s}$ variables to break the symmetry σ.

```
rup +1 t3 >= 1 : -1 22;
rup +1 t2 >= 1 : -1 20;
rup +1 t1 >= 1 : -1 18;
pol 11 x1 + s;
pol 13 x2 + s;
pol 15 x3 + s;
pol 11 ~x3 + s;
pol 13 ~x4 + s;
pol 15 ~x1 + s;
pol 16 136 + s;
pol 18 ~t1 3 * + 135 4 * + s;
pol 20 ~t2 3 * + 134 4 * + s;
pol 22 ~t3 3 * + 133 4 * + s;
del range 10 26;
del range 133 137;
```

By the end of this fragment, the net effect of symmetry breaking has been adding the following constraints to the derived set:

$$
\begin{aligned}
x_3 + s_1 &\geq 1 & x_4 + \overline{s_1} + s_2 &\geq 1 \\
x_1 + \overline{s_2} + s_3 &\geq 1 & \overline{x_1} + s_1 &\geq 1 \\
\overline{x_2} + \overline{s_1} + s_2 &\geq 1 & \overline{x_3} + \overline{s_2} + s_3 &\geq 1 \\
\overline{x_1} + x_3 &\geq 1 & \overline{x_2} + x_4 + \overline{s_1} &\geq 1 \\
x_1 + \overline{x_3} + \overline{s_2} &\geq 1 & x_2 + \overline{x_4} + \overline{s_3} &\geq 1
\end{aligned}
$$

Since these constraints do not affect the core set, we can still break another symmetry. The symmetry τ is larger than σ, but the proof proceeds through similar strokes. First, a symmetry breaker circuit is introduced through redundance-based strengthening.

```
red +1 ~s4 +1 x5 +1 ~x4 >= 1 : s4 -> 0;
red +2 s4 +1 ~x5 +1 x4 >= 2 : s4 -> 1;
red +3 ~s5 +2 s4 +1 x6 +1 ~x3 >= 3 : s5 -> 0;
red +2 s5 +2 ~s4 +1 ~x6 +1 x3 >= 2 : s5 -> 1;
red +3 ~s6 +2 s5 +1 x1 +1 ~x6 >= 3 : s6 -> 0;
red +2 s6 +2 ~s5 +1 ~x1 +1 x6 >= 2 : s6 -> 1;
red +3 ~s7 +2 s6 +1 x2 +1 ~x5 >= 3 : s7 -> 0;
red +2 s7 +2 ~s6 +1 ~x2 +1 x5 >= 2 : s7 -> 1;
red +3 ~s8 +2 s7 +1 x3 +1 ~x2 >= 3 : s8 -> 0;
red +2 s8 +2 ~s7 +1 ~x3 +1 x2 >= 2 : s8 -> 1;
red +1 ~t1 +1 ~x5 +1 x4 >= 1 : t1 -> 0;
red +2 t1 +1 x5 +1 ~x4 >= 2 : t1 -> 1;
red +4 ~t2 +3 t1 +1 ~s4 +1 ~x6 +1 x3 >= 4 : t2 -> 0;
red +3 t2 +3 ~t1 +1 s4 +1 x6 +1 ~x3 >= 3 : t2 -> 1;
red +4 ~t3 +3 t2 +1 ~s5 +1 ~x1 +1 x6 >= 4 : t3 -> 0;
red +3 t3 +3 ~t2 +1 s5 +1 x1 +1 ~x6 >= 3 : t3 -> 1;
red +4 ~t4 +3 t3 +1 ~s6 +1 ~x2 +1 x5 >= 4 : t4 -> 0;
```

```
red +3 t4 +3 ~t3 +1 s6 +1 x2 +1 ~x5 >= 3 : t4 -> 1;
red +4 ~t5 +3 t4 +1 ~s7 +1 ~x3 +1 x2 >= 4 : t5 -> 0;
red +3 t5 +3 ~t4 +1 s7 +1 x3 +1 ~x2 >= 3 : t5 -> 1;
red +4 ~t6 +3 t5 +1 ~s8 +1 ~x4 +1 x1 >= 4 : t6 -> 0;
red +3 t6 +3 ~t5 +1 s8 +1 x4 +1 ~x1 >= 3 : t6 -> 1;
```

Then, a symmetry breaking constraint is introduced through dominance-based strengthening.

```
dom +1 t6  >= 1 : x5 x4 x6 x3 x1 x6 x2 x5 x3 x2 x4 x1  : subproof
```

Then the first dominance proof goal is resolved.

```
scope leq
proofgoal #1
rup +1 ~$d6 >= 1;
rup >= 0;
pol 170;
pol 171;
rup +1 ~$a1 +1 $a1 >= 1;
rup +1 $a1 +1 ~$a1 >= 1;
pol 172 -2 2 * +;
pol 173 -2 2 * +;
rup +1 ~$a2 +1 $a2 >= 1;
rup +1 $a2 +1 ~$a2 >= 1;
pol 174 -2 2 * +;
pol 175 -2 2 * +;
rup +1 ~$a3 +1 $a3 >= 1;
rup +1 $a3 +1 ~$a3 >= 1;
pol 176 -2 2 * +;
pol 177 -2 2 * +;
rup +1 ~$a4 +1 $a4 >= 1;
rup +1 $a4 +1 ~$a4 >= 1;
pol 178 -2 2 * +;
pol 179 -2 2 * +;
rup +1 ~$a5 +1 $a5 >= 1;
rup +1 $a5 +1 ~$a5 >= 1;
rup >= 0;
pol 180;
pol 181;
rup +1 ~$d1 +1 $d1 >= 1;
rup +1 $d1 +1 ~$d1 >= 1;
pol 182 -2 3 * + -22 +;
pol 183 -2 3 * + -24 +;
rup +1 ~$d2 +1 $d2 >= 1;
rup +1 $d2 +1 ~$d2 >= 1;
pol 184 -2 3 * + -22 +;
pol 185 -2 3 * + -24 +;
rup +1 ~$d3 +1 $d3 >= 1;
rup +1 $d3 +1 ~$d3 >= 1;
pol 186 -2 3 * + -22 +;
pol 187 -2 3 * + -24 +;
rup +1 ~$d4 +1 $d4 >= 1;
rup +1 $d4 +1 ~$d4 >= 1;
pol 188 -2 3 * + -22 +;
pol 189 -2 3 * + -24 +;
rup +1 ~$d5 +1 $d5 >= 1;
rup +1 $d5 +1 ~$d5 >= 1;
pol 190 -2 3 * + -22 +;
pol 191 -2 3 * + -24 +;
rup +1 $d1 +1 ~s4 >= 1;
rup +1 $d2 +1 ~s5 >= 1;
```

```
rup +1 $d3 +1 ~s6 >= 1;
rup +1 $d4 +1 ~s7 >= 1;
rup +1 $d5 +1 ~s8 >= 1;
rup +1 t1 +1 ~$a1 >= 1;
rup +1 t2 +1 ~$a2 >= 1;
rup +1 t3 +1 ~$a3 >= 1;
rup +1 t4 +1 ~$a4 >= 1;
rup +1 t5 +1 ~$a5 >= 1;
rup +1 t2 +1 ~t1 +1 $d1 >= 1;
rup +1 t3 +1 ~t2 +1 $d2 >= 1;
rup +1 t4 +1 ~t3 +1 $d3 >= 1;
rup +1 t5 +1 ~t4 +1 $d4 >= 1;
rup +1 t6 +1 ~t5 +1 $d5 >= 1;
rup +1 $d2 +1 ~$d1 +1 t1 >= 1;
rup +1 $d3 +1 ~$d2 +1 t2 >= 1;
rup +1 $d4 +1 ~$d3 +1 t3 >= 1;
rup +1 $d5 +1 ~$d4 +1 t4 >= 1;
rup +1 $d6 +1 ~$d5 +1 t5 >= 1;
rup +1 $d1 +1 t1 >= 1;
rup +1 $d1 +1 t2 >= 1;
rup +1 $d2 +1 t1 >= 1;
rup +1 $d2 +1 t2 >= 1;
rup +1 $d2 +1 t3 >= 1;
rup +1 $d3 +1 t2 >= 1;
rup +1 $d3 +1 t3 >= 1;
rup +1 $d3 +1 t4 >= 1;
rup +1 $d4 +1 t3 >= 1;
rup +1 $d4 +1 t4 >= 1;
rup +1 $d4 +1 t5 >= 1;
rup +1 $d5 +1 t4 >= 1;
rup +1 $d5 +1 t5 >= 1;
rup +1 $d5 +1 t6 >= 1;
rup +1 $d6 +1 t5 >= 1;
rup +1 $d6 +1 t6 >= 1;
rup >= 1;
qed #1 : -1;
end scope;
```

The second proof goal follows, and finishes the dominance proof:

```
scope geq
proofgoal #2
rup +1 $d5 >= 1;
rup +1 $d4 >= 1;
rup +1 $d3 >= 1;
rup +1 $d2 >= 1;
rup +1 $d1 >= 1;
rup +1 ~s4 +1 $a1 >= 1;
rup +1 ~s5 +1 $a2 >= 1;
rup +1 ~s6 +1 $a3 >= 1;
rup +1 ~s7 +1 $a4 >= 1;
rup +1 ~s8 +1 $a5 >= 1;
rup +1 t1 >= 1;
rup +1 t2 >= 1;
rup +1 t3 >= 1;
rup +1 t4 >= 1;
rup +1 t5 >= 1;
rup >= 1;
qed #2 : -1;
end scope;
qed dom;
```

instance	variables	constraints	generators	support size
PHP(n)	O(n^2)	O(n^3)	O(n)	O(n)
RPHP(n)	O(n^2)	O(n^3)	O(n)	O(n)
ClqCl(n,6,5)	O(n^2)	O(n^2)	O(n)	O(n)
TseitinGrid(n)	O(n^2)	O(n^2)	O(n^2)	varies
Count(n,3)	O(n^3)	O(n^5)	O(n)	varies

Table 1: *Some data on the families of crafted benchmarks that we used: The number of variables and constraints of the instance, the number of generators that* SATSUMA*finds, and the support size of the symmetries that* SATSUMA*finds.*

Finally, some constraints are simplified.

```
rup +1 t5 >= 1 : -1 167;
rup +1 t4 >= 1 : -1 165;
rup +1 t3 >= 1 : -1 163;
rup +1 t2 >= 1 : -1 161;
rup +1 t1 >= 1 : -1 159;
pol 148 x5 + s;
pol 150 x6 + s;
pol 152 x1 + s;
pol 154 x2 + s;
pol 156 x3 + s;
pol 148 ~x4 + s;
pol 150 ~x3 + s;
pol 152 ~x6 + s;
pol 154 ~x5 + s;
pol 156 ~x2 + s;
pol 157 319 + s;
pol 159 ~t1 3 * + 318 4 * + s;
pol 161 ~t2 3 * + 317 4 * + s;
pol 163 ~t3 3 * + 316 4 * + s;
pol 165 ~t4 3 * + 315 4 * + s;
pol 167 ~t5 3 * + 314 4 * + s;
del range 147 171;
del range 314 320;
```

Through the second symmetry breaking proof, the following constraints are ultimately introduced:

$$x_4 + s_4 \geq 1 \qquad x_3 + \overline{s_4} + s_5 \geq 1$$
$$x_6 + \overline{s_5} + s_6 \geq 1 \qquad x_5 + \overline{s_6} + s_7 \geq 1$$
$$x_2 + \overline{s_7} + s_8 \geq 1 \qquad \overline{x_5} + s_4 \geq 1$$
$$\overline{x_6} + \overline{s_4} + s_5 \geq 1 \qquad \overline{x_1} + \overline{s_5} + s_6 \geq 1$$
$$\overline{x_2} + \overline{s_6} + s_7 \geq 1 \qquad \overline{x_3} + \overline{s_7} + s_8 \geq 1$$
$$x_4 + \overline{x_5} \geq 1 \qquad x_3 + \overline{x_6} + \overline{s_4} \geq 1$$
$$\overline{x_1} + x_6 + \overline{s_5} \geq 1 \qquad \overline{x_2} + x_5 + \overline{s_6} \geq 1$$
$$x_2 + \overline{x_3} + \overline{s_7} \geq 1 \qquad x_1 + \overline{x_4} + \overline{s_8} \geq 1$$

Appendix E Details on Crafted Benchmarks

In this appendix, we briefly describe the five crafted benchmark families that we used in our experimental evaluation. We use the standard notation $[n] = \{1, \ldots, n\}$.

Pigeonhole principle (PHP). The classical *pigeonhole principle* formula [Hak85] encodes the claim that n pigeons can fly into $n-1$ holes such that no two pigeons fly into the same hole. Clearly, this formula is UNSAT. The encoding has $n(n-1)$ variables x_{ij} for $i \in [n]$ and $j \in [n-1]$, where x_{ij} encodes that pigeon i flies into hole j. There are $(n-1)\binom{n}{2}$ binary clauses of the form $\overline{x_{ik}} \vee \overline{x_{jk}}$, for all $1 \leq i < j \leq n$ and $k \in [n-1]$, claiming that it is not the case that both pigeon i and pigeon j fly into hole k. Together, these clauses encode that no two pigeons fly into the same hole. In addition, there are n clauses of the form $\bigvee_{1 \leq j \leq n-1} x_{ij}$ for $i \in [n]$, encoding that pigeon i must fly into some hole. The symmetries of PHP are permuting the holes and permuting the pigeons. A set of generators for the corresponding symmetry group are $n-2$ symmetries that swap two holes and $n-1$ symmetries that swap two pigeons. Each such symmetry affects $O(n)$ variables.

Relativized pigeonhole principle (RPHP). The *relativized pigeonhole principle* formula [AMO13, ALN16] encodes the claim that n pigeons can fly into m resting places and then onwards to $n-1$ holes such that no two pigeons fly into the same resting place or the same hole. Formally, the RPHP formula claims that there exist maps $p \colon [n] \to [m]$ and $q \colon [m] \to [n-1]$ such that p is injective and q is injective on the range of p. Similarly to PHP, this formula is UNSAT. Here we choose $m = 2n$. Then the formula contains $4n^2$ variables and $O(n^3)$ constraints. The symmetries of RPHP are permuting the holes, permuting the pigeons, and permuting the resting places.

Clique-coloring formulas (ClqCl). The *clique-coloring* formula [Pud97] encodes the claim that there exists a graph on n vertices that contains a clique on k vertices and admits a coloring on c vertices. This formula is UNSAT for $k > c$. The formula contains $\binom{n}{2}$ variables to encode the edges of the graph, kn variables to encode a function from $[k]$ to $[n]$ representing the clique, and nc variables to encode a function from $[n]$ to $[c]$ representing the coloring. Here we choose $k = 6$ and $c = 5$. Then the formula has $O(n^2)$ constraints. The symmetries of the clique-coloring formula are the symmetries permuting the vertices of the graph.

Counting formulas (Count). The *counting* formula [PW85] encodes the claim that we can partition n elements into sets of size k each. The formula has $\binom{n}{k}$ variables, where each variable encodes that a particular set is chosen. There are $O(n^{2k-1})$ constraints, encoding that any two chosen sets must be disjoint. In our experiments, we choose $k = 3$. The symmetries of the counting formula are permuting the elements.

Tseitin formulas on a grid (Tseitin Grid). The *Tseitin formulas* [Tse68] consider a graph $G = (V, E)$ and a charge function $f \colon V \to \{0, 1\}$. The formula has a variable

for each edge, and the formula encodes that the XOR of the variables corresponding to the edges adjacent to some vertex v equals the charge $f(v)$. Here we consider the Tseitin formula on an $n \times n$ grid with even charge, i.e., $f(v) = 0$ for all $v \in V$. For each cycle in the graph G the Tseitin formula has a negation symmetry that flips all variables corresponding to edges in that cycle.

References

[ABR24] Markus Anders, Sofia Brenner, and Gaurav Rattan. Satsuma: Structure-based symmetry breaking in SAT. In *Proceedings of the 27th International Conference on Theory and Applications of Satisfiability Testing (SAT '24)*, volume 305 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 4:1–4:23, August 2024.

[ALN16] Albert Atserias, Massimo Lauria, and Jakob Nordström. Narrow proofs may be maximally long. *ACM Transactions on Computational Logic*, 17(3):19:1–19:30, May 2016. Preliminary version in *CCC '14*.

[AMO13] Albert Atserias, Moritz Müller, and Sergi Oliva. Lower bounds for DNF-refutations of a relativized weak pigeonhole principle. In *Proceedings of the 28th Annual IEEE Conference on Computational Complexity (CCC '13)*, pages 109–120, June 2013.

[AMS03] Fadi A. Aloul, Igor L. Markov, and Karem A. Sakallah. Shatter: efficient symmetry-breaking for boolean satisfiability. In *Proceedings of the 40th Annual Design Automation Conference (DAC '03)*, page 836–839. Association for Computing Machinery, June 2003.

[AW13] Tobias Achterberg and Roland Wunderling. Mixed integer programming: Analyzing 12 years of progress. In Michael Jünger and Gerhard Reinelt, editors, *Facets of Combinatorial Optimization*, pages 449–481. Springer, 2013.

[BBB+24] Suresh Bolusani, Mathieu Besançon, Ksenia Bestuzheva, Antonia Chmiela, João Dionísio, Tim Donkiewicz, Jasper van Doornmalen, Leon Eifler, Mohammed Ghannam, Ambros Gleixner, Christoph Graczyk, Katrin Halbig, Ivo Hedtke, Alexander Hoen, Christopher Hojny, Rolf van der Hulst, Dominik Kamp, Thorsten Koch, Kevin Kofler, Jurgen Lentz, Julian Manns, Gioni Mexi, Erik Mühmer, Marc E. Pfetsch, Franziska Schlösser, Felipe Serrano, Yuji Shinano, Mark Turner, Stefan Vigerske, Dieter Weninger, and Lixing Xu. The SCIP Optimization Suite 9.0. Technical report, Optimization Online, February 2024. Available at `https://optimization-online.org/2024/02/the-scip-optimization-suite-9-0/`.

[BGMN23] Bart Bogaerts, Stephan Gocht, Ciaran McCreesh, and Jakob Nordström. Certified dominance and symmetry breaking for combinatorial optimisation. *Journal of Artificial Intelligence Research*, 77:1539–1589, August 2023. Preliminary version in *AAAI '22*.

[BHvMW21] Armin Biere, Marijn J. H. Heule, Hans van Maaren, and Toby Walsh, editors. *Handbook of Satisfiability*, volume 336 of *Frontiers in Artificial Intelligence and Applications*. IOS Press, 2nd edition, February 2021.

[BN21] Samuel R. Buss and Jakob Nordström. Proof complexity and SAT solving. In Biere et al. [BHvMW21], chapter 7, pages 233–350.

[CCT87] William Cook, Collette Rene Coullard, and György Turán. On the complexity of cutting-plane proofs. *Discrete Applied Mathematics*, 18(1):25–38, November 1987.

[CHH+17] Luís Cruz-Filipe, Marijn J. H. Heule, Warren A. Hunt Jr., Matt Kaufmann, and Peter Schneider-Kamp. Efficient certified RAT verification. In *Proceedings of the 26th International Conference on Automated Deduction (CADE-26)*, volume 10395 of *Lecture Notes in Computer Science*, pages 220–236. Springer, August 2017.

[CS15] Geoffrey Chu and Peter J. Stuckey. Dominance breaking constraints. *Constraints*, 20(2):155–182, April 2015. Preliminary version in *CP '12*.

[DBB17] Jo Devriendt, Bart Bogaerts, and Maurice Bruynooghe. Symmetric explanation learning: Effective dynamic symmetry handling for SAT. In *Proceedings of the 20th International Conference on Theory and Applications of Satisfiability Testing (SAT '17)*, volume 10491 of *Lecture Notes in Computer Science*, pages 83–100. Springer, August 2017.

[DBBD16] Jo Devriendt, Bart Bogaerts, Maurice Bruynooghe, and Marc Denecker. Improved static symmetry breaking for SAT. In *Proceedings of the 19th International Conference on Theory and Applications of Satisfiability Testing (SAT '16)*, volume 9710 of *Lecture Notes in Computer Science*, pages 104–122. Springer, July 2016.

[GKL+05] Ian P. Gent, Tom Kelsey, Steve A. Linton, Iain McDonald, Ian Miguel, and Barbara M. Smith. Conditional symmetry breaking. In *Proceedings of the 11th International Conference on Principles and Practice of Constraint Programming (CP '05)*, volume 3709 of *Lecture Notes in Computer Science*, pages 256–270. Springer, October 2005.

[GMM+24] Stephan Gocht, Ciaran McCreesh, Magnus O. Myreen, Jakob Nordström, Andy Oertel, and Yong Kiam Tan. End-to-end verification for subgraph solving. In *Proceedings of the 38th AAAI Conference on Artificial Intelligence (AAAI '24)*, pages 8038–8047, February 2024.

[GN03] Evgueni Goldberg and Yakov Novikov. Verification of proofs of unsatisfiability for CNF formulas. In *Proceedings of the Conference on Design, Automation and Test in Europe (DATE '03)*, pages 886–891, March 2003.

[GN21] Stephan Gocht and Jakob Nordström. Certifying parity reasoning efficiently using pseudo-Boolean proofs. In *Proceedings of the 35th AAAI Conference on Artificial Intelligence (AAAI '21)*, pages 3768–3777, February 2021.

[Goc22] Stephan Gocht. *Certifying Correctness for Combinatorial Algorithms by Using Pseudo-Boolean Reasoning*. PhD thesis, Lund University, June 2022. Available at `https://portal.research.lu.se/en/publications/certifying-correctness-for-combinatorial-algorithms-by-using-pseu`.

[Hak85] Armin Haken. The intractability of resolution. *Theoretical Computer Science*, 39(2-3):297–308, August 1985.

[HHW15] Marijn J. H. Heule, Warren A. Hunt Jr., and Nathan Wetzler. Expressing symmetry breaking in DRAT proofs. In *Proceedings of the 25th International Conference on Automated Deduction (CADE-25)*, volume 9195 of *Lecture Notes in Computer Science*, pages 591–606. Springer, August 2015.

[IJ24] Markus Iser and Christoph Jabs. Global benchmark database. In *Proceedings of the 27th International Conference on Theory and Applications of Satisfiability Testing (SAT '24)*, volume 305 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 18:1–18:10, August 2024.

[JHB12] Matti Järvisalo, Marijn J. H. Heule, and Armin Biere. Inprocessing rules. In *Proceedings of the 6th International Joint Conference on Automated Reasoning (IJCAR '12)*, volume 7364 of *Lecture Notes in Computer Science*, pages 355–370. Springer, June 2012.

[KT24] Leszek Kołodziejczyk and Neil Thapen. The strength of the dominance rule. In *Proceedings of the 27th International Conference on Theory and Applications of Satisfiability Testing (SAT '24)*, volume 305 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 20:1–20:22, August 2024.

[LENV17] Massimo Lauria, Jan Elffers, Jakob Nordström, and Marc Vinyals. CNFgen: A generator of crafted benchmarks. In *Proceedings of the 20th International Conference on Theory and Applications of Satisfiability Testing (SAT '17)*, volume 10491 of *Lecture Notes in Computer Science*, pages 464–473. Springer, August 2017.

[PR19] Marc E. Pfetsch and Thomas Rehn. A computational comparison of symmetry handling methods for mixed integer programs. *Math. Program. Comput.*, 11(1):37–93, 2019.

[Pud97] Pavel Pudlák. Lower bounds for resolution and cutting plane proofs and monotone computations. *Journal of Symbolic Logic*, 62(3):981–998, September 1997.

[PW85] Jeff B. Paris and Alex J. Wilkie. Counting problems in bounded arithmetic. In *Methods in Mathematical Logic: Proceedings of the 6th Latin American Symposium on Mathematical Logic*, volume 1130 of *Lecture Notes in Mathematics*, pages 317–340. Springer, 1985.

[RM16] Olivier Roussel and Vasco M. Manquinho. Input/output format and solver requirements for the competitions of pseudo-Boolean solvers. Revision 2324. Available at `http://www.cril.univ-artois.fr/PB16/format.pdf`, January 2016.

[Sak21] Karem A. Sakallah. Symmetry and satisfiability. In Biere et al. [BHvMW21], chapter 13, pages 509–570.

[TMK+19] Yong Kiam Tan, Magnus O. Myreen, Ramana Kumar, Anthony C. J. Fox, Scott Owens, and Michael Norrish. The verified CakeML compiler backend. *Journal of Functional Programming*, 29:e2:1–e2:57, February 2019.

[Tse68] Grigori Tseitin. On the complexity of derivation in propositional calculus. In A. O. Silenko, editor, *Structures in Constructive Mathematics and Mathematical Logic, Part II*, pages 115–125. Consultants Bureau, New York-London, 1968.

[Urq99] Alasdair Urquhart. The symmetry rule in propositional logic. *Discrete Applied Mathematics*, 96–97:177–193, October 1999.

[Wal06] Toby Walsh. General symmetry breaking constraints. In *Proceedings of the 12th International Conference on Principles and Practice of Constraint Programming (CP '06)*, volume 4204 of *Lecture Notes in Computer Science*, pages 650–664. Springer, September 2006.

[WHH14] Nathan Wetzler, Marijn J. H. Heule, and Warren A. Hunt Jr. DRAT-trim: Efficient checking and trimming using expressive clausal proofs. In *Proceedings of the 17th International Conference on Theory and Applications of Satisfiability Testing (SAT '14)*, volume 8561 of *Lecture Notes in Computer Science*, pages 422–429. Springer, July 2014.

Proof Logging for Projected Enumeration (and Counting?) Problems in VeriPB

Abstract

When a certifying solver claims that a solution is optimal or that a problem is unsatisfiable, it demonstrates this convincingly by giving a proof log which can be checked by an independent (and ideally formally-verified) proof checker. Such an approach should also be viable for enumeration problems ("I have listed all solutions explicitly") and counting problems ("there are exactly 42 solutions"), but the currently most popular proof logging systems contain several vital features which are incompatible with this goal. We explain how the VeriPB system can be modified for enumeration and counting proofs whilst retaining as much as possible of its powerful "strengthening" and "deletion" features. We implement this extension both inside VeriPB's user-friendly proof checker and elaborator, and the formally verified CakePB backend, and use this to obtain formally verified enumerations of solutions for a range of constraint solving and graph problem instances.

1 Introduction

Modern solvers for combinatorial and automated reasoning problems are able to solve many large and challenging problem instances quickly, but their complexity means that sometimes bugs lead to incorrect answers being produced. One popular remedy to this issue is to make solvers *certifying*: that is, to have them output a mathematical *proof log* detailing how a conclusion was reached. This proof log can then be checked independently by a proof-checking tool, and ideally this tool will be sufficiently simple that it can be formally verified. Proof logging is a standard feature for Boolean satisfiability (SAT) solvers, and is now being adopted in other application areas that use richer data types and constraints.

Ciaran McCreesh, Jakob Nordström, Andy Oertel, and Yong Kiam Tan. "Proof Logging for Projected Enumeration (and Counting?) Problems in VeriPB". *in review*.

In the early SAT proof logging systems operating on formulas in conjunctive normal form (CNF), a proof was, in effect, a sequence of clausal constraints, each of which could easily be checked to be implied by (that is, to be a logical consequence of) previous constraints in the problem or proof. In such a system, each step preserves satisfiability of the original problem, and so if we can derive an unsatisfiable constraint (such as an empty clause) then we have shown that the original problem has no solutions. Ideally, a proof system should also make it easy for solver authors to generate proofs: for example, for a basic conflict-driven clause learning (CDCL) SAT solver, several proof systems have been designed such that a proof can simply be the list of inferred clauses, in the order they are derived [GN03, HKB17, JHB12].

For more general problems, such as those with non-Boolean variables or with non-clausal constraints, two kinds of approach have been proposed. The first involves extending the proof format with new rules directly corresponding to every kind of constraint and inference used by every individual solver (e.g. [BBC+23, FSM+24, GCS17, VS10]). The second instead adds only a few simple but very general rules to the proof format, in the hope that these are powerful enough to express all of the reasoning carried out by many different solvers. The VeriPB system we discuss in this paper adopts this latter approach, and it has been used to certify a range of constraint programming techniques [GMN22, MM23, MM25, MMN24], dynamic programming [DMM+24], graph solvers [GMM+20, GMM+24, GMN20], advanced SAT and MaxSAT solvers [BBN+23, GN21, IOT+24, VDB22], pseudo-Boolean solvers [KLM+25], and classical planning [DHN+25]. Critically, although VeriPB itself operates with 0–1 linear inequalities, or *pseudo-Boolean (PB) constraints*, the system can be used to certify solvers that carry out general constraint reasoning, without requiring these solvers to modify their reasoning or to be aware of the underlying pseudo-Boolean representation when solving. To enable this, VeriPB proof logs can include explicit derivations of constraints, which can be used to express non-trivial inferences, e.g., all-different consequences [EGMN20], and other complex forms of reasoning, without requiring new bespoke rules in the proof format.

In practice, solver users do not just care about unsatisfiable decision problem instances. Demonstrating *satisfiability* for NP-complete problems is not difficult: by definition, these problems will have an easily checkable witness. For proofs of optimality for single-objective optimisation problems, we can combine these two concepts as we explain below. However, for problems where the user expects an *enumeration* (an explicit list of all solutions) or a *counting* (to be told how many solutions there are, without requiring an explicit list), the situation is more intricate. These kinds of problems arise in areas such as computer algebra, graph theory, combinatorics, and physics (e.g. [AMV22, CMPS19, DJKK12, HAJ25, IC20, KS24, TTT06, VBV25]), and are the topic of this paper. In the following section, we will explain why enumeration proofs are not straightforward. We then describe an extension to the VeriPB system (including its formally-verified CakePB checker) that supports these proofs, evaluate its implementation on a range of problem instances, and finally introduce further extensions for preprocessing and for counting problems.

2 Proof Logging Beyond Implicational Proofs of Unsatisfiability

Extending a simple implicational proof system to support optimisation can be relatively simple, if the proof system has some natural way of representing integer inequalities (which many proof systems, including those based upon CNF, do not). One way would be to add a "solution-improving" step to the proof language. Such a step would take a witness, which the proof checker would validate, and would be equivalent to introducing a new constraint saying that the objective function must beat the objective value for that witness. Here we would be certifying optimality by proving that "there is a solution with objective value X, and the instance is unsatisfiable if you want an objective value better than X".

A similar scheme could be used for enumeration proofs: we could add a "solution-witnessing" proof step, which would introduce a blocking constraint, and then a proof deriving unsatisfiability would be saying "there are no solutions, other than all of the ones listed". Indeed, the first version of VeriPB supported a limited form of such proofs, and this was used to check results for maximal clique enumeration problem instances that were misreported in the literature [GMM+20].

An immediate limitation of this kind of enumeration proof is that there may be more solutions to an encoded problem seen by a solver or proof checker than there are to the real-world problem the user actually cares about. This can occur when using an encoding which is not *parsimonious*, or which is not *arc consistent* in the SAT sense of the word. For some encodings, this can be solved by *projection* [GKS09]: informally, by defining a subset of variables which we will call the *preserved set*, and by asking for the number of unique solutions with respect to the restriction onto that set. Inside a proof, the blocking constraint would then be introduced only with respect to those variables (although for soundness, the witnessing assignment must still demonstrate satisfiability over all constraints, including those using variables outwith the preserved set). It turns out that dealing with projection will be helpful for the remainder of this paper, even for problems where there is a well-behaved encoding, and so we will assume we are always working in a projected setting (where the preserved set is "all of the variables" for simple enumeration problems).

So with this established, why do proofs for enumeration remain a problem? It turns out that most practical proof systems [WHH14, HKB19], including newer versions of VeriPB [BGMN23, GN21], have two additional features which are both vital for practical performance and efficiency, but which make a simple blocking-based approach unsound: strengthening, and deletions.

2.1 Strengthening

Many practical solvers for combinatorial optimisation and automated reasoning problems do not use purely "implicational" reasoning. That is, they derive constraints that are satisfied by *some* solutions, but not necessarily *all* solutions. As a simple example, with *pure literal elimination*, a SAT solver may observe that a literal only appears in positive form in all clauses [DLL62]. If we are solving a decision problem, we can infer that such a literal can be set to true unconditionally,

because if a solution exists to the problem then a solution necessarily exists where that literal is set to true. More generally, we might want to introduce symmetry-elimination constraints [AW13, GSVW14, Sak21]: for example, if our entire problem is $X + Y = 1$ over integer variables X and Y with identical domains, then it is safe to consider only the case where $X \leq Y$. These inferences may exclude *some* solutions, but will never turn a satisfiable instance into an unsatisfiable one.

For this reason, most modern proof logging formats weaken the invariant preserved by proof steps to being some notion like "without loss of satisfiability", rather than "implied". We will use the term "strengthening" to refer to proof steps that only give this weaker invariant. The exact form and power of these rules varies heavily between systems. However, the general intuition behind most of these rules is that they allow us to introduce a constraint C if we can show that, given an assignment α which would satisfy the problem but does not satisfy C, then we can somehow *patch* α to an assignment where we can easily verify that it would also satisfy C, whilst still satisfying the remaining constraints [BT21, GN21, JHB12]. This patching rule is either given explicitly (such as by listing a set of substitutions), or is implicit in how the rule is defined, and must allow a proof checker to verify the patching conditions efficiently. For example, in our toy symmetry breaking example, we would show that $X \leq Y$ is safe because if we are given a solution where instead $X > Y$, we can patch this by swapping the values of X and Y. This notion also extends to optimisation problems, where we can reason "without loss of optimality" at each step and must verify that the patching procedure does not worsen the objective [BGMN23]. Clearly, strengthening of this kind cannot be used in enumeration proofs, because it allows a proof to exclude some solutions to the problem.

A further common use of strengthening is to introduce fresh variables during a proof. For example, suppose a proof author wants to introduce a *reification* constraint C saying $f \rightarrow (X + Y \geq 2)$, where the variable f is fresh. This is a valid thing to do, because if we are given a solution to the original problem which violates C, then we can "patch" this solution by setting f to false. Having done this, we could then also introduce a constraint C' saying $f \leftarrow (X + Y \geq 2)$, because we could patch a bad solution for C' by setting f to true (and note that this also satisfies C).

It turns out that the ability to introduce fresh variables makes proof systems much stronger theoretically [Kra19, Tse68], and is vital in practice for efficient proofs of certain kinds of inference [DMM$^+$24, GMN22]. However, fresh variables also cause problems for enumeration proofs. For example, introducing one direction of a reification might introduce new solutions, leading to overcounting (but it is hard to verify whether a given rule application *definitely* introduces new solutions or not). We might hope that, if we restrict ourselves to only introducing both directions of a reification simultaneously, then we would avoid this trap (at the expense of reasoning power): we discuss this further in what follows.

2.2 Deletions

A second vital feature of most proof systems is deletions [HHW14]. We are often working with solvers that do an exponential amount of work overall, but do not

keep most of this work in memory. For example, in a simple backtracking algorithm, once a solver has backtracked from a certain depth, it no longer needs to remember anything about the subtrees it explored below that depth. Similarly, solvers based around learning schemes such as CDCL will periodically "forget" most of the clauses they have learned, in part to avoid memory exhaustion [MSLM21]. It is extremely helpful if a proof can indicate "I promise the remainder of the proof will not use the following constraints again, so if you are checking this proof you do not need to keep these constraints in memory".

Clearly, an unconditional deletion rule would allow non-solutions to be claimed as solutions, since a devious proof logger could simply delete all input constraints and claim that an arbitrary assignment is a "solution". Note, however, that this is not a problem if our proof system only allows for proofs of unsatisfiability, and has a separate external mechanism if we wish to certify satisfiability (for example, DRAT [WHH14] operates this way).

In a purely implicational proof system, deletions can be tamed by forbidding any constraint in the input from being deleted (although this is more restrictive than necessary). We could extend this to also forbid deleting solution-blocking constraints, which would allow for enumeration proofs, but this would come at the expense of requiring the proof checker to keep all solution-blocking constraints in memory rather than deleting them upon backtracking.

When combined with strengthening rules, deletions become even more complicated. For example, a problem may exhibit *conditional symmetries*, and we may wish to write a proof that first shows that the condition always holds, and then deletes parts of the input, allowing us to introduce a symmetry elimination constraint. The VeriPB proof system allows this when it can be shown to be sound, through a mechanism which we discuss in Section 3.

2.3 Enumeration Proofs with Strengthening and Deletions

Having now established why enumeration proofs are not straightforward, in the next section we will show the exact conditions needed to allow strengthening (and deletions) whilst preserving enumeration counts in the VeriPB proof format. It turns out that projection will be vital: the key observation is that, when combined with projection, the strengthening rule preserves equienumerability so long as the "patching" procedure does not touch any variable in the preserved set. Additionally, we show that VeriPB's existing support for deletions in optimisation problems can also be adapted to work for projected enumeration. Subsequently, we show how the proof system can allow the preserved set to be altered mid-proof, and that doing so provides interesting ways of generating proofs for solvers that count solutions in a way which is more efficient than simple enumeration.

We implement these features both in the developer-friendly VeriPB checker and elaborator, and in the formally-verified CakePB checker [ABB+26, GMM+24, IOT+24, KLM+25]. As part of this, proofs of all three of the theorems that will follow have been formally verified inside a proof assistant. Interestingly, the approach we describe is *not* suitable for other popular proof formats, which would need a fundamental redesign to support a similar feature.

3 Projected Enumeration Proofs in VeriPB

We will briefly review some basics about pseudo-Boolean (PB) reasoning and the VeriPB proof system, where we refer the reader to the VeriPB documentation [ABB+25] and to Bogaerts et al. [BGMN23] for a full description of the VeriPB proof system, and to Buss and Nordström [BN21] for more details on pseudo-Boolean reasoning. A *literal* ℓ of a *Boolean variable* x with domain 0 and 1 is either x itself of its negation $\overline{x} = 1 - x$. We often refer to 0 as *false* and 1 as *true*. A *pseudo-Boolean constraint* is a 0–1 integer linear inequality $\sum_i a_i \ell_i \geq A$ over literals ℓ_i, where in addition we can assume *normalised form* so that all integers a_i and A are positive and literals are over distinct variables. We note that for a PB constraint $C = \sum_i a_i \ell_i \geq A$ with $W = \sum_i a_i$ the negation $\neg C$ is another PB constraint $\sum_i a_i \ell_i \leq A - 1$ or $\sum_i a_i \overline{\ell_i} \geq W - A + 1$ in normalised form; the reification of C for a fresh variable f can be expressed with $f \rightarrow C \equiv A\overline{f} + \sum_i a_i \ell_i \geq A$ and $f \leftarrow C \equiv \overline{f} \rightarrow \neg C$.

Let F be a *pseudo-Boolean formula*, which is a conjunction $\bigwedge_i C_i$ of pseudo-Boolean constraints C_i. A *(partial) assignment* is a (partial) mapping σ from variables to $\{0, 1\}$, which is extended to literals by $\sigma(\overline{x}) = 1 - \sigma(x)$. A *solution* to F is an assignment σ such that each constraint in the formula is satisfied, i.e., $\sum_{\sigma(\ell_i)=1} a_i \geq A$ (again assuming normalised form). A pseudo-Boolean constraint C *propagates* a literal ℓ under a partial assignment σ if C is falsified by ℓ mapped to 0 and σ, i.e., $\sum_{\sigma(\ell_i)\neq 0 \wedge \ell_i \neq \ell} a_i < A$, where σ can be extended by mapping ℓ to 1. A constraint C is implied by *reverse unit propagation (RUP)* from a formula F if iteratively propagating constraints $F \wedge \neg C$ and updating an assignment σ results in a falsified constraint, i.e., such that $\sum_{\sigma(\ell_i)\neq 0} a_i < A$.

Given a *preserved set* of variables P, and denoting a restriction of the domain of a function by $|$, we say that the set $\{\sigma|_P : \sigma \text{ is a solution for } F\}$ is the *solution set of F projected onto P*. An *enumeration problem* is to explicitly list all the elements of this set exactly once each, whilst a *counting problem* is to determine the cardinality of this set. The *blocking constraint* for a given solution σ with respect to P is the constraint $\sum_{p \in P : \sigma(p)=0} p + \sum_{p \in P : \sigma(p)=1} \overline{p} \geq 1$, which says that at least one variable in P has to take a different value than in σ.

To support enumeration, we have extended VeriPB to allow its pseudo-Boolean input formula to specify a preserved set of variables; if projection is not desired, this set can be every variable appearing in the formula. This is similar to how optimisation is handled, where the input formula can come with an additional objective expression to minimise.

Example 1. Consider the following pseudo-Boolean problem instance, which we specify in the extended OPB format [RM16] supported by our modified version of VeriPB. To read this file, note that tilde denotes negation, and that each inequality is a weighted linear sum—here, each inequality is an "at least one" constraint. The inequality each line corresponds to is given on the right-hand side.

`preserved: x1 x2 ;`	preserved set initialized to $\{x_1, x_2\}$
`1 x1 1 x2 1 x3 >= 1 ;`	$x_1 + x_2 + x_3 \geq 1$
`1 x1 1 x2 1 x4 >= 1 ;`	$x_1 + x_2 + x_4 \geq 1$
`1 ~x1 1 ~x2 >= 1 ;`	$\overline{x}_1 + \overline{x}_2 \geq 1$

```
1 ~x1 1 ~x2 1 ~x3 >= 1 ;          x̄1 + x̄2 + x̄3 ≥ 1
1 x3 1 x4 >= 1 ;                  x3 + x4 ≥ 1
```

A little thought shows that there are seven solutions to the problem without projection, but with projection onto x_1 and x_2, there are only three.

example continues below. . .

We will now walk through a VeriPB enumeration proof. Most steps in a proof will introduce a new pseudo-Boolean constraint, based upon what is known so far. Initially the only things we know are the constraints in the input, which are taken as axioms.

Example 2 (continued)**.** Continuing with our example above, we might want to start our proof by observing that x_4 is a pure literal, and so can always be set to true without changing the number of projected solutions. We use a redundance strengthening rule to do this, with the patching routine being to swap x_4 with its negation. Recall that this means that "if you give me a solution which satisfies every constraint except $x_4 \geq 1$, then by negating x_4, i.e., by making it true, you can clearly see that we will now have a solution which satisfies every constraint, and also $x_4 \geq 1$". In the following VeriPB proof examples, the percent sign starts a comment.

```
pseudo-Boolean proof version 3.0
@purex4  red 1 x4 >= 1 : x4 -> ~x4 ; % x4 is pure
```

The second line starts with a `@label`, which gives a name to a constraint so we can refer to it later; constraints also automatically have numerical identifiers, but this is harder for a human to follow. *example continues below. . .*

For optimisation problems, the VeriPB proof system has the proof rule denoted `soli`, for "solution improving", which takes a set of literals L as its argument. This rule says "check that L unit propagates to a solution α where every variable in the objective is assigned, and then introduce an objective-improving constraint $f \leq f(\alpha) - 1$ for this solution". For enumeration, we have introduced a similar rule `solx`, for "solution excluding", which checks that its argument literals unit propagate to a solution where every variable in the preserved set is assigned, and then introduces the blocking constraint for this solution with respect to the preserved set.

Example 3 (continued)**.** Suppose now we find that $\{x_1, \overline{x}_2, \overline{x}_3\}$ is a solution. We can log this as follows.

```
@sol1    solx x1 ~x2 ~x3 ;           % log our first solution
```

Note here that we *do* need to specify an assignment for x_3, because it could be set either way, but x_4's value is obviously forced by the previous constraint. We also give a name to the blocking constraint that this proof step creates (which will be $\overline{x}_1 + x_2 \geq 1$).

Once we have this blocking constraint, it is clear that there are no more solutions where x_1 is true, so we can introduce the constraint $x_1 \leq 0$ (or $\overline{x}_1 \geq 1$ in normalised form) via reverse unit propagation. Having done so, we will never need the blocking constraint again, and so we can delete it; the `core` line in between will be explained below.

```
@x1false rup 1 ~x1 >= 1 ;   % no more solutions with x1 true
         core id @x1false ; % ...and move this to core
         del id @sol1 ;     % no need to keep subtree constraints
```

Looking at what remains, x_3 is effectively a pure literal, because the fourth input constraint is the only place where it appears negatively, and this will always be satisfied by setting x_1 to false. After that, we can log the two solutions where x_1 is false, and use the two new blocking constraints to learn that x_1 must be true. We can then delete the blocking constraints.

```
@purex3  red 1 x3 >= 1 : x3 -> ~x3 ; % now x3 is effectively pure
@sol2    solx ~x1 x2 ; % log our second solution
@sol3    solx ~x1 ~x2 ; % and our third solution
@x1true  rup 1 x1 >= 1 ; % no more solutions with x1 false
         core id @x1true ; % ...and move this to core
         % no need to keep subtree constraints
         del id @sol2 @sol3 ;
```

At this point, we have shown that x_1 must be both true and false, which is enough to establish a contradiction by reverse unit propagation. So, we can learn $0 \geq 1$, and then delete both constraints that talk about x_1, and for good measure we no longer care that x_3 and x_4 are pure either.

```
@contra  rup >= 1 ;                % backtrack to root node
         core id @contra ;         % ...and move this to core
         % no need to keep these constraints
         del id @x1false @x1true ;
         del id @purex3 @purex4 ;  % no need to remember purity
```

Now all that remains is to conclude that we have actually enumerated three solutions to completeness. *example continues below...*

A VeriPB proof ends with an output section and conclusion. For enumeration problems, we add support for three kinds of conclusion. The first is `ENUMERATION_COMPLETE` n : i, which states that we have witnessed and obtained blocking constraints for exactly n different projected solutions, and then proved unsatisfiability with constraint i. The second is `ENUMERATION_PARTIAL` n, which states that we have witnessed n different projected solutions, but that we do not claim to have found them all. The third is to extend the existing `NONE` conclusion with an `output EQUIENUMERABLE` option, which we discuss in the following section: this allows us to write proofs that alter a formula whilst preserving the number of solutions.

Example 4 (continued)**.** We can finish our proof with the following:

```
output NONE ;
conclusion ENUMERATION_COMPLETE 3 : @contra ;
end pseudo-Boolean proof ;
```

This is now a complete proof, which is accepted by our enhanced VeriPB and CakePB checkers. ◻

With this example completed, we will now go into more detail on why our modified proof system remains sound. We will only sketch proofs of the formal claims that follow—one reason why this is in order is that formally machine-verified proofs of all of these claims are part of the CakePB backend. Firstly, we observe that our new `solx` rule allows us to count solutions if we are only using the implicational parts of the proof system.

Proposition 1. *If a proof uses only implicational rules and* `solx`*, then each* `solx` *rule invocation excludes exactly one new projected solution.*

Proof sketch. Since the proof only uses implicational rules, the set of solutions does not change. By definition, the `solx` rule checks that the claimed solution is in fact valid. Then, we cannot log the same solution (under projection) twice, because the second time we attempt it, our witness will violate the blocking constraint introduced by the first attempt. □

We now need to establish that none of the remaining proof rules allow us to alter the number of solutions. The two non-implicational rules in the VeriPB proof system are strengthening and deletions.

Proposition 2. *Applications of strengthening where the patching procedure does not touch the preserved set always preserve the projected solution set.*

Proof sketch. It is most illuminating to look at the two ways we have seen strengthening be used, and to see why they do not break this property.

For strengthening rules that introduce an implication under a fresh variable, the patching procedure only touches the fresh variable, which is not inside the preserved set. This new constraint does not exclude solutions, since the fresh variable can be assigned to satisfy the constraint. This rule may introduce additional solutions, but the fresh variable is not included in the preserved set, so these additional solutions will be equivalent under projection.

For strengthening rules that touch existing variables which are not in the preserved set, this may exclude (unprojected) solutions. However, the patching procedure leaves the value of the variables in the preserved set unchanged, so that the patched solution restricted to the preserved set is the same.

Note that neither of these arguments depend upon the detail of exactly how we justify the patching routine, and so they are valid for both redundance, and the more general dominance strengthening rules [BGMN23]. Observe also that we are able to introduce both half- and two-way reifications on fresh variables this way, but as a special case of the more general result, rather than needing an explicitly restricted proof rule. □

To understand deletions, we need to explain a little more about VeriPB's *checked deletions* mode. Within a proof, every constraint is considered to be either *core* or *derived*. Initially, everything in the input goes into the core set, whilst constraints inside the proof go into either the core set or the derived set depending upon how they are created. Additionally, any derived constraint can be moved into core by an explicit proof step. Constraints can always be deleted from the derived set, but to delete a constraint C from the core set, we must show that C can be

rederived using redundance strengthening by just using other core constraints. Critically, once a constraint is in the core set, either it or stronger constraints remain there for the remainder of the proof. This system was developed to allow for optimisation proofs that use symmetry or dominance reasoning, and we refer readers to Bogaerts et al. [BGMN23] for full details.

Proposition 3. *If all deletions are checked, then deletion will not allow a proof to claim non-solutions with respect to the preserved variables as a solution.*

Proof sketch. Since we show that the deleted constraint can be rederived by redundance strengthening from the other constraints in the core set, the core set with and without the deleted constraint have the same projected solution set by Proposition 2. □

The VeriPB proof system handles the solution-improving constraints for optimisation problems by putting them into the core set. We do the same for enumeration.

Proposition 4. *If all deletions are checked, and if blocking constraints are created in the core set, then deletions will not allow projected solutions to be counted more than once.*

Proof sketch. If we are able to delete a blocking constraint from the core, then it must be possible to rederive it from the other core constraints, since all deletions are checked. So the blocking constraint always remains a consequence of the core set by Proposition 3. □

At this point, we wish to claim the following.

Theorem 5. *If a VeriPB proof uses only checked deletions, does not strengthen using any patching procedure which touches a preserved variable, has blocking constraints created in the core set, and is able to derive a contradiction constraint, then the number of projected solutions to the original problem is precisely the number of times the* `solx` *rule is used.*

To do this, we are effectively claiming "there are no other elements of the VeriPB proof system that we have forgotten about that could break things". The most convincing demonstration we have that this is true is that we have formally verified the entire proof system inside CakePB, including the conclusions supported.

3.1 Formal Verification of Enumeration Proofs

CakePB is a formally verified proof checker that can handle a subset of the VeriPB proof rules. To support an end-to-end certified solving workflow, VeriPB operates in *elaboration* mode, where it takes a proof written in a solver-friendly *augmented* proof format and elaborates it to a simpler *kernel* format proof for CakePB to check. Intuitively, elaboration adds proof details which are easy to fill with an unverified search, while the kernel format contains a set of primitive pseudo-Boolean proof rules that can be efficiently implemented and formally verified in a proof assistant with reasonable effort.

We have extended the formal verification of CakePB (in the HOL4 proof assistant [SN08]) to support all of the new rules described in this paper. Let

us recall CakePB's refinement-based verification methodology [GMM+24], and describe our extensions at each step.

1. The formalisation starts by defining the pseudo-Boolean problem semantics, and then defines the abstract PB proof system. We extended the semantics to support projected enumeration, namely:

   $$\begin{aligned} &\mathsf{sem_concl}\ \mathit{pbf}\ \mathit{obj}\ \mathit{pres}\ (\mathsf{EEnum}\ n\ \mathit{complete}) \stackrel{\text{def}}{=} \\ &\quad n \le \mathsf{card}\ (\mathsf{proj_pres}\ \mathit{pres}\ \{\ w \mid \mathsf{satisfies}\ w\ \mathit{pbf}\ \}) \land \\ &\quad (\mathit{complete} \Rightarrow \mathsf{card}\ (\mathsf{proj_pres}\ \mathit{pres}\ \{\ w \mid \mathsf{satisfies}\ w\ \mathit{pbf}\ \}) \le n) \end{aligned}$$

   Here, we are defining the semantics of conclusions (sem_concl) that can be made about a PB problem with constraints *pbf*, objective *obj*, and preserved set *pres*. For the new conclusion type EEnum we allow a proof to claim either partial or complete enumeration of n projected solutions. The proj_pres function projects the set of satisfying solutions for *pbf*, $\{w \mid \mathsf{satisfies}\ w\ \mathit{pbf}\}$, onto *pres*; card is the cardinality of a set in HOL.

2. The next step is to implement the kernel proof rules with an abstract proof checking algorithm. In this phase, we introduce the `solx` command (among others), and we also modify existing rules to be compatible with the preserved set. In essence, we have a machine-checked proof of Theorem 5 (and its constituent propositions) showing that whenever the checker successfully processes a proof with a formal conclusion (like EEnum), then that conclusion is indeed true of the input PB problem.

3. In the third phase, the abstract proof checker is refined into an optimized machine-code implementation using CakeML [GMKN17, MO14, TMK+19]. Thanks to the relatively small-scale change to the proof system, the updates here were straightforward. We ultimately obtain an end-to-end proof that the machine-code semantics (either x64 or ARMv8) is sound for our updated PB problem semantics.

The steps above result in what we refer to as the *backend* CakePB implementation for pseudo-Boolean problems. As discussed earlier, pseudo-Boolean proofs can also be used more broadly to certify solvers for more general combinatorial and automated reasoning paradigms. For this, we need an additional step to set up a formally verified *frontend* which ensures that the original (non-pseudo-Boolean) problem is re-encoded into pseudo-Boolean format correctly. This yields a certification flow where a general combinatorial solver solves problems with its native constraint reasoning, while emitting pseudo-Boolean justifications that are checked by VeriPB and CakePB to establish valid claims about the input problem.

4. (Optional step) For a given combinatorial problem, e.g., maximal clique enumeration [TTT06], we start by formalising the problem semantics as follows.

$$\begin{aligned}
\mathsf{is_clique}\ vs\ (v,e) &\overset{\text{def}}{=} vs \subseteq \mathsf{count}\ v\ \wedge \\
&\quad\ \forall x\ y.\ x \in vs \wedge y \in vs \wedge x \neq y \Rightarrow \mathsf{is_edge}\ e\ x\ y \\
\mathsf{is_maximal_clique}\ vs\ g &\overset{\text{def}}{=} \mathsf{is_clique}\ vs\ g\ \wedge \\
&\quad\ \forall vs'.\ \mathsf{is_clique}\ vs'\ g \wedge vs \subseteq vs' \Rightarrow vs = vs' \\
\mathsf{maximal_cliques}\ g &\overset{\text{def}}{=} \{\ vs \mid \mathsf{is_maximal_clique}\ vs\ g\ \}
\end{aligned}$$

Here, a graph g is represented as a pair consisting of a natural number v for vertices labeled count $v \overset{\text{def}}{=} \{0, \ldots, v-1\}$ and edges e (looked up by the is_edge function). As usual, the set vs forms a clique if it is a set of vertices in g that mutually share an edge. A clique vs satisfies is_maximal_clique iff there is no larger clique containing vs.

Then, we prove the soundness for an encoder into PB:

$$\begin{aligned}
&\vdash \mathsf{good_graph}\ g \wedge \mathsf{enc}\ g = (pres,pbf) \wedge \\
&\quad \mathsf{sem_concl}\ pbf\ \mathsf{None}\ pres\ (\mathsf{EEnum}\ n\ b) \Rightarrow \\
&\quad\quad \mathsf{if}\ b\ \mathsf{then}\ \mathsf{card}\ (\mathsf{maximal_cliques}\ g) = n\ \mathsf{else}\ n \leq \mathsf{card}\ (\mathsf{maximal_cliques}\ g)
\end{aligned}$$

This theorem says that running the problem encoder enc on the input graph g produces a preserved set *pres* and set of PB constraints *pbf* such that, if we can show sem_concl holds with an enumeration conclusion for this PB problem, then that conclusion correctly reports the number of enumerated maximal cliques. Here, enc uses the direct encoding for cliques where a Boolean variable x_i is true iff vertex i is in the chosen clique; the None argument to sem_concl indicates no objective function. Putting this verified frontend encoding together with the verified backend PB proof checker CakePB yields an end-to-end verified proof checker for maximal clique enumeration which we call CakePBmclique.

3.2 Experiments for Enumeration Proofs

We carried out two sets of experiments to validate our implementation. Firstly, we adapted the maximal clique enumeration implementation of Tomita et al.'s algorithm [TTT06] by Gocht et al. [GMM+20] to use the VeriPB 3 proof format, to use our new `solx` command, projection, and conclusion, and added support for deletions (which would not have been sound for enumeration problems in the original VeriPB 1 proof format). We then successfully replicated all of Gocht et al.'s clique enumeration experiments using both VeriPB and CakePBmclique, except for one instance where CakePBmclique exceeded a 48GByte RAM limit when verifying the encoding.

Secondly, we adapted the Glasgow Constraint Solver's test suite [GMN22]. Part of the testing process for this solver is to create a large number of random constraint satisfaction problems, verify that the solver generates the same set of solutions as an extremely simple generate-and-test routine, and then to use VeriPB to check an enumeration proof for the same problem. Previously, these proofs relied upon underspecified solution-excluding behaviour as a workaround, and

could not formally claim a complete enumeration as a conclusion or make use of CakePB. Our extensions to VeriPB rectify this, and additionally allow us to pass the proofs through CakePB for formal verification. Additionally, we altered the solver to use projection to explicitly exclude auxiliary variables in the encoding from the solution count, meaning that its tests now guarantee that solution counts are in terms of the high-level constraint programming variables, not the low-level encoding. With these additions, all of the solver's tests still pass the stricter verification setup, and we have formally verified the results of tens of thousands of enumeration proofs for constraint satisfaction problems.

4 Proofs for Preprocessing and Counting Without Enumerating

Instead of checking a complete solving process, VeriPB (and CakePB) can also be used to check that a preprocessor preserves certain guarantees [IOT+24]. In order to make claims about the properties of a rewritten formula produced by a preprocessor or presolver, proofs use an *output section* with a heading such as `output EQUISATISFIABLE FILE`, followed by an empty *conclusion section* with the heading `conclusion NONE`. This particular output section heading tells VeriPB to check that the proof demonstrates that a pseudo-Boolean problem instance provided in a file named on the command line is *equisatisfiable* with respect to the original pseudo-Boolean problem instance (i.e., the output formula is unsatisfiable if and only if the input formula is unsatisfiable).

We have extended VeriPB and its proof system to also support `output EQUIENUMERABLE` as an output type. This feature can be used to check that two pseudo-Boolean problem instances have the same number of (projected) solutions, without actually listing those solutions. We do *not* require that these two pseudo-Boolean problem instances have the same preserved set, and instead allow variables to be added to and removed from the preserved set during the proof, so long as a justification is given—we explain how this is possible below.

Beyond preprocessing, this feature has a second potential use. In some situations, a user may only want to know how many solutions there are to a problem instance, without requiring an explicit list. In such cases, sometimes solvers can produce results more efficiently than carrying out an enumeration. One family of techniques to do this involves recompiling the problem into a new structure which is efficiently countable, such as some kind of decision diagram [Dar04, YM25]. Another is to decompose the problem, enumerate the solutions to each decomposition, and then use an inclusion-exclusion argument to generate the count [Kar82, Rya23, Rys63].

We remark that specialized certification methods are available for both exact [FHR22] and approximate [TYS+24] model counting. Of these, the CPOG framework [BNAH23] similarly enables a proof of equivalence between an input CNF and the output of a knowledge compiler. Our VeriPB-based approach could offer a generic means of certifying transformation-based tools.

Example 5. Suppose someone wishes to convince you that there are exactly three (projected) solutions to our previous example. It should be sufficient for them to provide a VeriPB proof that ends

```
output EQUIENUMERABLE FILE ;
conclusion NONE ;
end pseudo-Boolean proof ;
```

where the output file provided could be

```
preserved: s1 s2 s3 ;
1 s1 1 s2 1 s3 = 1 ;            exactly one s variable is true
```

Why is this? Recall that our original problem is entirely over x_i variables. Here, our output file is now over a fresh set of s_i variables, where we have a constraint saying that exactly one of these three variables is true. We do not have any other constraints and so clearly there are three solutions to this problem. *example continues below...*

This example on its own is probably too simple to convincingly illustrate the benefits of an alternative approach. However, suppose our output constraints only said that $\sum_{i=1}^{10} s_i = 1$, $\sum_{i=1}^{5} t_i = 1$, $\sum_{i=1}^{8} u_i = 1$, and $s_1 \to \overline{t}_1$. From this we could easily and efficiently determine that there are $(10 \times 5 \times 8) - (1 \times 8) = 392$ solutions using an inclusion-exclusion argument, with a proof that could potentially be of the order $10 + 5 + 8 + 1$ steps long. Such an approach mirrors how some subgraph-counting algorithms work [Rya23]. We will now outline how such proofs could be produced (for the simpler example, although this technique generalises cleanly), together with the additional features we have implemented to make this possible.

4.1 Tabulating Solutions as Extension Variables

Rather than logging solutions using `solx`, our proofs will introduce an extension variable s_1 for the first solution found, reifying the negation of the blocking constraint that `solx` would introduce (resembling a proof for autotabulation [GMN22]). We will then move the forward direction of this implication to the core.

Example 6 (continued)**.** We might start a proof as follows. The first `red` line uses strengthening for a pure literal, as before, and the second and third `red` lines introduce the reification variable.

```
pseudo-Boolean proof version 3.0
@purex4   red 1 x4 >= 1 : x4 -> ~x4 ;          % x4 is pure
          % log our first solution
@sol1f    red 2 ~s1 1 x1 1 ~x2 >= 2 : s1 -> 0 ;
          % ...in both directions
@sol1b    red 1 s1  1 ~x1 1 x2 >= 1 : s1 -> 1 ;
          core id @sol1f ;
```

example continues below...

Going forward, for every subsequent constraint we generate, we include $\overline{s}_1$ as an additional assumption, until we hit a second solution. We then introduce a second extension variable s_2 for this solution, and change our assumption to be $\overline{s}_1 \wedge \overline{s}_2$, and so on.

Example 7 (continued)**.** So, our proof can continue like this.

```
@x1false rup 1 s1 1 ~x1 >= 1 ; % unless s1, x1 is false
@purex3  red 1 s1 1 x3 >= 1 : x3 -> ~x3 ; % unless s1, x3 is pure
         % log our second solution
@sol2f   red 2 ~s2 1 ~x1 1 x2 >= 2 : s2 -> 0 ;
         % ...in both directions
@sol2b   red 1 s2  1 x1 1 ~x2 >= 1 : s2 -> 1 ;
         core id @sol2f ;
         % log our third solution
@sol3f   red 2 ~s3 1 ~x1 1 ~x2 >= 2 : s3 -> 0 ;
         % ...in both directions
@sol3b   red 1 s3  1 x1 1 x2 >= 1 : s3 -> 1 ;
         core id @sol3f ;
@x1true  rup 1 s1 1 s2 1 s3 1 x1 >= 1 ; % otherwise, x1 is true
@al1sol  rup 1 s1 1 s2 1 s3 >= 1 ; % no other solutions
         core id @al1sol ;
         del id @x1true @x1false ;
```

example continues below...

We then continue our solving process until completion, at which point, instead of deriving contradiction, we have derived an at-least-one constraint over a set of s_i variables exactly corresponding to the number of solutions found. We move this to the core, and can now delete any remaining constraints from the search process.

Next, we want to turn this at-least-one constraint into an exactly-one constraint. This can be done efficiently using a standard technique in VeriPB proofs, by first deriving a not-both constraint for each pair of s_i variables: we refer to Gocht et al. [GMM+20] for an explanation. We also move this constraint to core.

Example 8 (continued)**.** In this case, we can generate the at-most-one constraint using the following sequence of steps, where the reverse unit propagation succeeds because the s_i variables represent different projected solutions.

```
@s1ors2  rup 1 ~s1 1 ~s2 >= 1 ; % s1 and s2 are exclusive
@s1ors3  rup 1 ~s1 1 ~s3 >= 1 ; % s1 and s3 are exclusive
@s2ors3  rup 1 ~s2 1 ~s3 >= 1 ; % s2 and s3 are exclusive
@am1sol  pol -3 -2 + -1 + 2 d ; % recover the at-most-one
         del id @s1ors2 @s1ors2 @s1ors3 ; % intermediate steps
         core id @am1sol ;
```

example continues below...

At this point, our core set should consist precisely of all the original constraints, all the forward implications for our s_i variables, and the exactly-one constraint over the s_i variables. Our preserved set remains unchanged, which is what we will address next.

4.2 Extending the Preserved Set

We would like to be able to add our s_i variables to the preserved set. By careful thought, in this particular case, we can see that doing so would not change the number of solutions. To convince the proof checker of this, we need an argument such as the following.

Theorem 6. *Given a pseudo-Boolean formula F with preserved set P, a variable v, and some constraint C only over variables in P, if at some point during a* VeriPB *proof it is possible to derive the constraints $v \rightarrow C$ and $v \leftarrow C$ using only core constraints, then adding v to P for the remainder of the proof will not affect the projected enumeration count.*

Proof sketch. Any solution to F projected on to P will either satisfy or falsify C, since C only contains variables in P, and thus v's value will be uniquely determined by this solution. Furthermore, because both directions of the reification are derivable from the core set, this property cannot be altered by any later step in the proof. □

We have therefore extended VeriPB with the proof rule `preserved_add` that takes a variable s and an associated constraint C over preserved variables, for which it can be proven that $s \leftrightarrow C$ holds. Just as for other advanced VeriPB rules, there is the possibility to provide explicit subproofs that show why this claim holds. However, our example here is simple enough that VeriPB will be able to figure out all required details with so-called *auto-proving* without any help from the proof logger, and so we instead refer the interested reader to the VeriPB technical documentation for more details on how subproofs work.

Example 9 (continued)**.** Each of our s_i variables is in fact defined by an if-and-only-if reification, so we can use this as the constraint.

```
% add s1 to preserved set
preserved_add s1 1 x1 1 ~x2 >= 2 ;
% add s2 to preserved set
preserved_add s2 1 ~x1 1 x2 >= 2 ;
% add s3 to preserved set
preserved_add s3 1 ~x1 1 ~x2 >= 2 ;
```

example continues below...

4.3 Shrinking the Preserved Set

Our next challenge is to remove the x_i variables from the preserved set. Again, we need a general justification of why it is safe to do this.

Theorem 7. *Given a pseudo-Boolean formula F with preserved set P, a preserved variable $v \in P$, and some constraint C only over the variables in $P \setminus \{v\}$, if at some point during a* VeriPB *proof it is possible to derive the constraints $v \rightarrow C$ and $v \leftarrow C$ using only core constraints, then removing v from P for the remainder of the proof will not affect the projected enumeration count.*

Proof sketch. This holds since any solution to F projected on to P will either satisfy or falsify C, which will force v to 0 or 1. So all these conditions together mean that the value of v will always be uniquely determined by any assignment to the remaining variables in P. □

We have implemented this through the `preserved_rm` rule, which takes a variable and a constraint given explicitly. (Interestingly, this operation is reversible: a proof checker could in principle go over the proof backwards to efficiently recover

solutions in terms of the original variables, without the need for a separate solution reconstruction stack.)

Example 10 (continued)**.** We can obtain the desired constraint by looking at precisely which s_i variables assign a variable to true.

```
% remove x1 from preserved set
preserved_rm x1 1 s1 >= 1 ;
% remove x2 from preserved set
preserved_rm x2 1 s2 >= 1 ;
```

example continues below. . .

4.4 Deleting Remaining Constraints

At this point, we have a convincing proof that there are *at most* three solutions, but we still retain all the original constraints over the x_i variables in the core set, so it is not clear that each s_i assignment would actually give a valid solution. (Nor is this fact trivially checkable, since the s_i variables do not force values for x_3 or x_4.) The final step, then is to delete each original constraint in turn.

To delete a constraint from core, we must show that the constraint can be reconstructed using only the other remaining core constraints. We should always be able to do this, using a fairly simple procedure. Suppose first that we were not producing a projection proof, and that our original preserved set contained every x_i variable. Consider any original constraint C. We will use a proof by contradiction, showing that the negation of C together with the reification constraints for each s_i variable implies unsatisfiability, as follows. For each s_i variable in turn, if that variable is true, then it implies an assignment of x_i variables that satisfy every original constraint, including C, which means that the negation of C is falsified. Then, we resolve over the at-least-one s_i constraint to reach our overall contradiction.

When we do have a preserved set, this procedure is potentially slightly more complicated. In our example, the constraint $x_3 + x_4 \geq 1$ is not contradicted by assigning true to any s_i variable, and we relied upon (conditional) pure literal reasoning to handle these variables. This is not an insurmountable problem: we can solve this by temporarily moving the pure literal constraints into core, and then deleting them at the end of the proof using a substitution justification (we refer to the VeriPB documentation [ABB+25] for details, since this feature is the same as for optimisation proofs).

Example 11 (continued)**.** In our example, the proof by contradiction is sufficiently simple in each case that it can be autoproved, so we only need the following.

```
core id @purex3 @purex4 ;
del id 1 2 3 4 5 ;
del id @purex3 : x3 -> ~x3 ;
del id @purex4 : x4 -> ~x4 ;
```

Finally, we can delete the forward implications of the s_i-variables using their corresponding solution for the patching procedure.

```
del id @sol1f : x1 -> 1 x2 -> 0 ;
del id @sol2f : x1 -> 0 x2 -> 1 ;
del id @sol3f : x1 -> 0 x2 -> 0 ;
```

At this point, we can terminate the proof using the desired output statement. □

This approach generalises to outputs over multiple sets of variables, potentially with further restrictions between them. With these new proof rules in place, we have enough reasoning power to handle certain inclusion-exclusion arguments. If we wished to develop a complete and general end-to-end verification system, we would also need to agree upon an efficiently-countable but expressive subset of pseudo-Boolean formulae, and to implement a formally-verified frontend for this subset. Choosing an appropriate and useful language would require a detailed study of commonly used decomposition and counting techniques.

5 Conclusion

Enhancing combinatorial solvers with *proof logging* to make them *certifying* is currently the most successful way of ensuring that such solvers produce trustworthy results. Proof logging technology is by now mature for decision problems in Boolean satisfiability (SAT) solving, and is also starting to be developed for more and more optimisation and automated reasoning paradigms. However, the most widely adopted proof logging systems are inherently unable to support proof logging for problems such as enumerating or counting the number of solutions. Instead, proof logging support for such reasoning problems has so far only been available in bespoke tools with limited applicability.

In this work, we demonstrated how the VeriPB proof logging system, which has previously been shown to provide a unified proof logging format for a wide range of combinatorial solving and optimisation problems, can also be extended to support also solution enumeration proofs in the same proof format. In contrast to previous approaches, for VeriPB this is possible without sacrificing any of the advanced features that make the proof system so powerful, such as strengthening rules and checked deletion. This opens up the future possibility of also providing efficient proofs for counting problems (where it is crucial to not have to enumerate all solutions), by supporting proofs of solution-count-preserving transformations for the input formula to a format in which counting solutions is trivial.

References

[ABB+25] Markus Anders, Bart Bogaerts, Benjamin Bogø, Arthur Gontier, Wietze Koops, Ciaran McCreesh, Magnus O. Myreen, Jakob Nordström, Andy Oertel, Adrián Rebola-Pardo, and Yong Kiam Tan. Documentation of VeriPB and CakePB for the SAT competition 2025. Available at `https://satcompetition.github.io/2025/output.html`, April 2025.

[ABB+26] Markus Anders, Bart Bogaerts, Benjamin Bogø, Arthur Gontier, Wietze Koops, Ciaran McCreesh, Magnus O. Myreen, Jakob Nordström, Andy Oertel, Adrián Rebola-Pardo, and Yong Kiam Tan. Faster certified symmetry breaking using orders with auxiliary variables. In *Proceedings of the 40th AAAI Conference on Artificial Intelligence (AAAI '26)*, January 2026. To appear.

[AMV22] Özgür Akgün, Martín Mereb, and Leandro Vendramin. Enumeration of set-theoretic solutions to the yang-baxter equation. *Math. Comput.*, 91(335):1469–1481, 2022.

[AW13] Tobias Achterberg and Roland Wunderling. Mixed integer programming: Analyzing 12 years of progress. In Michael Jünger and Gerhard Reinelt, editors, *Facets of Combinatorial Optimization*, pages 449–481. Springer, 2013.

[BBC+23] Haniel Barbosa, Clark Barrett, Byron Cook, Bruno Dutertre, Gereon Kremer, Hanna Lachnitt, Aina Niemetz, Andres Nötzli, Alex Ozdemir, Mathias Preiner, Andrew Reynolds, Cesare Tinelli, and Yoni Zohar. Generating and exploiting automated reasoning proof certificates. *Communications of the ACM*, 66(10):86—95, October 2023.

[BBN+23] Jeremias Berg, Bart Bogaerts, Jakob Nordström, Andy Oertel, and Dieter Vandesande. Certified core-guided MaxSAT solving. In *Proceedings of the 29th International Conference on Automated Deduction (CADE-29)*, volume 14132 of *Lecture Notes in Computer Science*, pages 1–22. Springer, July 2023.

[BGMN23] Bart Bogaerts, Stephan Gocht, Ciaran McCreesh, and Jakob Nordström. Certified dominance and symmetry breaking for combinatorial optimisation. *Journal of Artificial Intelligence Research*, 77:1539–1589, August 2023. Preliminary version in *AAAI '22*.

[BHvMW21] Armin Biere, Marijn J. H. Heule, Hans van Maaren, and Toby Walsh, editors. *Handbook of Satisfiability*, volume 336 of *Frontiers in Artificial Intelligence and Applications*. IOS Press, 2nd edition, February 2021.

[BN21] Samuel R. Buss and Jakob Nordström. Proof complexity and SAT solving. In Biere et al. [BHvMW21], chapter 7, pages 233–350.

[BNAH23] Randal E. Bryant, Wojciech Nawrocki, Jeremy Avigad, and Marijn J. H. Heule. Certified knowledge compilation with application to verified model counting. In *Proceedings of the 26th International Conference on Theory and Applications of Satisfiability Testing (SAT '23)*, volume 271 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 6:1–6:20, July 2023.

[BT21] Sam Buss and Neil Thapen. DRAT and propagation redundancy proofs without new variables. *Logical Methods in Computer Science*, 17(2):12:1–12:31, April 2021. Preliminary version in *SAT '19*.

[CMPS19] Michael Codish, Alice Miller, Patrick Prosser, and Peter J. Stuckey. Constraints for symmetry breaking in graph representation. *Constraints An Int. J.*, 24(1):1–24, 2019.

[Dar04] Adnan Darwiche. New advances in compiling CNF to decomposable negation normal form. In *Proc. of ECAI*, pages 328–332, 2004.

[DHN+25] Simon Dold, Malte Helmert, Jakob Nordström, Gabriele Röger, and Tanja Schindler. Pseudo-Boolean proof logging for optimal classical planning. In *Proceedings of the 35th International Conference on Automated Planning and Scheduling (ICAPS '25)*, pages 54–63, November 2025.

[DJKK12] Andreas Distler, Christopher Jefferson, Tom W. Kelsey, and Lars Kotthoff. The semigroups of order 10. In Michela Milano, editor, *Principles and Practice of Constraint Programming - 18th International Conference, CP 2012, Québec City, QC, Canada, October 8-12, 2012. Proceedings*, volume 7514 of *Lecture Notes in Computer Science*, pages 883–899. Springer, 2012.

[DLL62] Martin Davis, George Logemann, and Donald Loveland. A machine program for theorem proving. *Communications of the ACM*, 5(7):394–397, July 1962.

[DMM+24] Emir Demirović, Ciaran McCreesh, Matthew McIlree, Jakob Nordström, Andy Oertel, and Konstantin Sidorov. Pseudo-Boolean reasoning about states and transitions to certify dynamic programming and decision diagram algorithms. In *Proceedings of the 30th International Conference on Principles and Practice of Constraint Programming (CP '24)*, volume 307 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 9:1–9:21, September 2024.

[EGMN20] Jan Elffers, Stephan Gocht, Ciaran McCreesh, and Jakob Nordström. Justifying all differences using pseudo-Boolean reasoning. In *Proceedings of the 34th AAAI Conference on Artificial Intelligence (AAAI '20)*, pages 1486–1494, February 2020.

[FHR22] Johannes Klaus Fichte, Markus Hecher, and Valentin Roland. Proofs for propositional model counting. In Kuldeep S. Meel and Ofer Strichman, editors, *25th International Conference on Theory and Applications of Satisfiability Testing, SAT 2022, Haifa, Israel, August 2-5, 2022*, volume 236 of *LIPIcs*, pages 30:1–30:24. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2022.

[FSM+24] Maarten Flippo, Konstantin Sidorov, Imko Marijnissen, Jeff Smits, and Emir Demirovic. A multi-stage proof logging framework to certify the correctness of CP solvers. In Paul Shaw, editor, *30th International Conference on Principles and Practice of Constraint Programming, CP 2024, Girona, Spain, September 2-6, 2024*, volume 307 of *LIPIcs*, pages 11:1–11:20. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2024.

[GCS17] Graeme Gange, Geoffrey Chu, and Peter J. Stuckey. Certifying optimality in constraint programming. Unpublished manuscript, 2017.

[GKS09] Martin Gebser, Benjamin Kaufmann, and Torsten Schaub. Solution enumeration for projected boolean search problems. In Willem-Jan van Hoeve and John N. Hooker, editors, *Integration of AI and OR Techniques in Constraint Programming for Combinatorial Optimization Problems*, pages 71–86, Berlin, Heidelberg, 2009. Springer Berlin Heidelberg.

[GMKN17] Armaël Guéneau, Magnus O. Myreen, Ramana Kumar, and Michael Norrish. Verified characteristic formulae for CakeML. In *Proceedings of the 26th European Symposium on Programming (ESOP '17)*, volume 10201 of *Lecture Notes in Computer Science*, pages 584–610. Springer, April 2017.

[GMM+20] Stephan Gocht, Ross McBride, Ciaran McCreesh, Jakob Nordström, Patrick Prosser, and James Trimble. Certifying solvers for clique and maximum common (connected) subgraph problems. In *Proceedings of the 26th International Conference on Principles and Practice of Constraint Programming (CP '20)*, volume 12333 of *Lecture Notes in Computer Science*, pages 338–357. Springer, September 2020.

[GMM+24] Stephan Gocht, Ciaran McCreesh, Magnus O. Myreen, Jakob Nordström, Andy Oertel, and Yong Kiam Tan. End-to-end verification for subgraph solving. In *Proceedings of the 38th AAAI Conference on Artificial Intelligence (AAAI '24)*, pages 8038–8047, February 2024.

[GMN20] Stephan Gocht, Ciaran McCreesh, and Jakob Nordström. Subgraph isomorphism meets cutting planes: Solving with certified solutions. In *Proceedings of the 29th International Joint Conference on Artificial Intelligence (IJCAI '20)*, pages 1134–1140, July 2020.

[GMN22] Stephan Gocht, Ciaran McCreesh, and Jakob Nordström. An auditable constraint programming solver. In *Proceedings of the 28th International Conference on Principles and Practice of Constraint Programming (CP '22)*, volume 235 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 25:1–25:18, August 2022.

[GN03] Evgueni Goldberg and Yakov Novikov. Verification of proofs of unsatisfiability for CNF formulas. In *Proceedings of the Conference on Design, Automation and Test in Europe (DATE '03)*, pages 886–891, March 2003.

[GN21] Stephan Gocht and Jakob Nordström. Certifying parity reasoning efficiently using pseudo-Boolean proofs. In *Proceedings of the 35th AAAI Conference on Artificial Intelligence (AAAI '21)*, pages 3768–3777, February 2021.

[GSVW14] Maria Garcia de la Banda, Peter J. Stuckey, Pascal Van Hentenryck, and Mark Wallace. The future of optimization technology. *Constraints*, 19(2):126–138, April 2014.

[HAJ25] Ruth Hoffmann, Özgür Akgün, and Christopher Jefferson. Composable constraint models for permutation enumeration. *Discrete Mathematics & Theoretical Computer Science*, vol. 26:1, Permutation Patterns 2023, Jan 2025.

[HHW14] Marijn J. H. Heule, Warren A. Hunt, and Nathan Wetzler. Bridging the gap between easy generation and efficient verification of unsatisfiability proofs. *Softw. Test. Verif. Reliab.*, 24(8):593–607, September 2014.

[HKB17] Marijn J. H. Heule, Benjamin Kiesl, and Armin Biere. Short proofs without new variables. In *Proceedings of the 26th International Conference on Automated Deduction (CADE-26)*, volume 10395 of *Lecture Notes in Computer Science*, pages 130–147. Springer, August 2017.

[HKB19] Marijn J. H. Heule, Benjamin Kiesl, and Armin Biere. Encoding redundancy for satisfaction-driven clause learning. In *Proceedings of the 25th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS '19)*, volume 11427 of *Lecture Notes in Computer Science*, pages 41–58. Springer, April 2019.

[IC20] Avraham Itzhakov and Michael Codish. Incremental symmetry breaking constraints for graph search problems. In *The Thirty-Fourth AAAI Conference on Artificial Intelligence, AAAI 2020, The Thirty-Second Innovative Applications of Artificial Intelligence Conference, IAAI 2020, The Tenth AAAI Symposium on Educational Advances in Artificial Intelligence, EAAI 2020, New York, NY, USA, February 7-12, 2020*, pages 1536–1543. AAAI Press, 2020.

[IOT+24] Hannes Ihalainen, Andy Oertel, Yong Kiam Tan, Jeremias Berg, Matti Järvisalo, Magnus O. Myreen, and Jakob Nordström. Certified MaxSAT preprocessing. In *Proceedings of the 12th International Joint Conference on Automated Reasoning (IJCAR '24)*, volume 14739 of *Lecture Notes in Computer Science*, pages 396–418. Springer, July 2024.

[JHB12] Matti Järvisalo, Marijn J. H. Heule, and Armin Biere. Inprocessing rules. In *Proceedings of the 6th International Joint Conference on Automated Reasoning (IJCAR '12)*, volume 7364 of *Lecture Notes in Computer Science*, pages 355–370. Springer, June 2012.

[Kar82] Richard M. Karp. Dynamic programming meets the principle of inclusion and exclusion. *Operations Research Letters*, 1(2):49–51, 1982.

[KLM+25] Wietze Koops, Daniel Le Berre, Magnus O. Myreen, Jakob Nordström, Andy Oertel, Yong Kiam Tan, and Marc Vinyals. Practically feasible proof logging for pseudo-Boolean optimization. In *Proceedings of*

the 31st International Conference on Principles and Practice of Constraint Programming (CP '25), volume 340 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 21:1–21:27, August 2025.

[Kra19] Jan Krajíček. *Proof Complexity*, volume 170 of *Encyclopedia of Mathematics and Its Applications*. Cambridge University Press, March 2019.

[KS24] Markus Kirchweger and Stefan Szeider. SAT modulo symmetries for graph generation and enumeration. *ACM Trans. Comput. Log.*, 25(3):1–30, 2024.

[MM23] Matthew McIlree and Ciaran McCreesh. Proof logging for smart extensional constraints. In *Proceedings of the 29th International Conference on Principles and Practice of Constraint Programming (CP '23)*, volume 280 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 26:1–26:17, August 2023.

[MM25] Matthew McIlree and Ciaran McCreesh. Certifying bounds propagation for integer multiplication constraints. In *Proceedings of the 39th AAAI Conference on Artificial Intelligence (AAAI '25)*, pages 11309–11317, February-March 2025.

[MMN24] Matthew McIlree, Ciaran McCreesh, and Jakob Nordström. Proof logging for the circuit constraint. In *Proceedings of the 21st International Conference on the Integration of Constraint Programming, Artificial Intelligence, and Operations Research (CPAIOR '24)*, volume 14743 of *Lecture Notes in Computer Science*, pages 38–55. Springer, May 2024.

[MO14] Magnus O. Myreen and Scott Owens. Proof-producing translation of higher-order logic into pure and stateful ML. *Journal of Functional Programming*, 24(2–3):284–315, January 2014.

[MSLM21] Joao Marques-Silva, Ines Lynce, and Sharad Malik. Conflict-driven clause learning SAT solvers. In Biere et al. [BHvMW21], chapter 13, pages 509–570.

[RM16] Olivier Roussel and Vasco M. Manquinho. Input/output format and solver requirements for the competitions of pseudo-Boolean solvers. Revision 2324. Available at `http://www.cril.univ-artois.fr/PB16/format.pdf`, January 2016.

[Rya23] Jessica Laurette Ryan. *Parameterised algorithms for counting subgraphs, matchings, and monochromatic partitions*. PhD thesis, University of Glasgow, 2023.

[Rys63] Herbert John Ryser. *Combinatorial mathematics*, volume 14. American Mathematical Soc., 1963.

[Sak21] Karem A. Sakallah. Symmetry and satisfiability. In Biere et al. [BHvMW21], chapter 13, pages 509–570.

[SN08] Konrad Slind and Michael Norrish. A brief overview of HOL4. In *Theorem Proving in Higher Order Logics, 21st International Conference, TPHOLs 2008, Montreal, Canada, August 18-21, 2008. Proceedings*, pages 28–32, 2008.

[TMK+19] Yong Kiam Tan, Magnus O. Myreen, Ramana Kumar, Anthony C. J. Fox, Scott Owens, and Michael Norrish. The verified CakeML compiler backend. *Journal of Functional Programming*, 29:e2:1–e2:57, February 2019.

[Tse68] Grigori Tseitin. On the complexity of derivation in propositional calculus. In A. O. Silenko, editor, *Structures in Constructive Mathematics and Mathematical Logic, Part II*, pages 115–125. Consultants Bureau, New York-London, 1968.

[TTT06] Etsuji Tomita, Akira Tanaka, and Haruhisa Takahashi. The worst-case time complexity for generating all maximal cliques and computational experiments. *Theor. Comput. Sci.*, 363(1):28–42, 2006.

[TYS+24] Yong Kiam Tan, Jiong Yang, Mate Soos, Magnus O. Myreen, and Kuldeep S. Meel. Formally certified approximate model counting. In Arie Gurfinkel and Vijay Ganesh, editors, *Computer Aided Verification - 36th International Conference, CAV 2024, Montreal, QC, Canada, July 24-27, 2024, Proceedings, Part I*, volume 14681 of *Lecture Notes in Computer Science*, pages 153–177. Springer, 2024.

[VBV25] Daimy Van Caudenberg, Bart Bogaerts, and Leandro Vendramin. Incremental sat-based enumeration of solutions to the yang-baxter equation. In Arie Gurfinkel and Marijn Heule, editors, *Tools and Algorithms for the Construction and Analysis of Systems - 31st International Conference, TACAS 2025, Held as Part of the International Joint Conferences on Theory and Practice of Software, ETAPS 2025, Hamilton, ON, Canada, May 3-8, 2025, Proceedings, Part II*, volume 15697 of *Lecture Notes in Computer Science*, pages 3–22. Springer, 2025.

[VDB22] Dieter Vandesande, Wolf De Wulf, and Bart Bogaerts. QMaxSATpb: A certified MaxSAT solver. In *Proceedings of the 16th International Conference on Logic Programming and Non-monotonic Reasoning (LPNMR '22)*, volume 13416 of *Lecture Notes in Computer Science*, pages 429–442. Springer, September 2022.

[VS10] Michael Veksler and Ofer Strichman. A proof-producing CSP solver. In *Proceedings of the 24th AAAI Conference on Artificial Intelligence (AAAI '10)*, pages 204–209, July 2010.

[WHH14] Nathan Wetzler, Marijn J. H. Heule, and Warren A. Hunt Jr. DRAT-trim: Efficient checking and trimming using expressive clausal proofs. In *Proceedings of the 17th International Conference on Theory and Applications of Satisfiability Testing (SAT '14)*, volume 8561 of *Lecture Notes in Computer Science*, pages 422–429. Springer, July 2014.

[YM25] Suwei Yang and Kuldeep S. Meel. Towards projected and incremental pseudo-boolean model counting. In Toby Walsh, Julie Shah, and Zico Kolter, editors, *AAAI-25, Sponsored by the Association for the Advancement of Artificial Intelligence, February 25 - March 4, 2025, Philadelphia, PA, USA*, pages 11399–11407. AAAI Press, 2025.