

universität ulm

Unleashing the Potential of Configuration Counting for Product-Line Analyses

Chico Sundermann
Born in Reiskirchen, Germany

Supervised by
Prof. Dr. Thomas Thüm

A cumulative dissertation submitted in fulfillment
of the requirements for the degree of

Dr. rer. nat.

at the Institute of Software Engineering and Programming Languages
Faculty of Engineering, Computer Science and Psychology

2025

Doctoral Thesis

Institute of Software Engineering and Programming Languages

Doctoral Thesis in Computer Science
Date of PhD Graduation: August 12th, 2025
Institution: Ulm University, Germany

Examiners

- Prof. Dr. Thomas Thüm
  Institute of Software Engineering and Automotive Informatics
  TU Braunschweig, Germany
- Prof. Dr. Jacobo Torán
  Institute of Theoretical Computer Science
  Ulm University, Germany
- Prof. Dr. Mathieu Acher
  Research Institute of Computer Science and Random Systems (IRISA)
  University of Rennes and INSA Rennes, France

Acting Dean

- Prof. Dr. Matthias Tichy
  Institute of Software Engineering and Programming Languages
  Ulm University, Germany

© 2025 Chico Sundermann
Unleashing the Potential of Configuration Counting for Product-Line Analyses

This work is licenced under the *Creative Commons Attribution 4.0* license. To view a copy of this license, visit https://creativecommons.org/licenses/by/4.0/ or send a letter to Creative Commons, PO Box 1866, Mountain View, CA 94042, USA.

Abstract

Product lines are part of most areas in our daily lives: cars, smartphones, personal computers, operating systems, and many more everyday commodities are highly customizable allowing users to select from hundreds or thousands of features to derive their tailored products. This large variability introduces various challenges for product-line engineers, management, and even users. One crucial challenge is understanding and managing the complexity of dependencies between features that disallow certain combinations of features (i.e., configurations) in final products. To tackle this complexity, a plethora of automated analyses have been proposed.

Over the recent years, several analyses emerged in literature and industry that depend on *configuration counting* (i.e., computing the number of valid configurations). These counting-based analyses span over the whole product-line development cycle supporting different stakeholders in understanding the configurability. For instance, unexpected changes in the configuration count can be used to detect design errors or the number of valid configurations containing a certain feature can be used to prioritize for resource allocation. Despite the consideration of configuration counting in research and practice over several domains, we identified three substantial hurdles for its applicability in practice which we tackle in this thesis: (1) knowledge on the possibilities facilitated by configuration counting is limited, (2) scalability in practice is largely unknown, and (3) as revealed by our work, scalable solutions for many applications are missing and demanded.

Even though some applications of configuration counting have been considered in research and practice, they are scattered across various publications, and computation instructions are often missing. Furthermore, within industry collaborations, we employed various novel applications in practice. As our first major contribution, we provide an overview of applications facilitated by configuration counting and classify the kinds of counting problems the applications depend on. Covering applications from literature and practice, our survey indicates the relevance of configuration counting for product-line engineering. However, configuration counting bases on the computationally complex problem of model counting (aka. #SAT). Even worse, our survey reveals that most applications rely on numerous counting operations. For systems in practice, some applications may require thousands or even millions of counting operations. Thus, to employ the identified applications, scalable solutions are required.

The performance of available solutions for configuration counting in practice is largely unknown. As our second major contribution, we examine the scalability of state-of-the-art strategies for configuration counting. In particular, we empirically evaluate the performance of 21 #SAT solvers on computing the number of valid configurations of industrial product lines. The evaluation reveals that available #SAT solvers scale to most industrial systems, but even the fastest solvers require a few seconds for many instances, which exceeds acceptable runtimes for applications depending on many counting operations.

The limited scalability of available solutions for applications relying on myriads of queries hinder adoption in practice. With our third contribution, we tackle the scalability for the multitude of counting queries. Our key idea for accelerating many counting queries employs knowledge compilation, which translates the dependencies between features into a format that

enables more efficient counting queries. Here, we collect and compare various knowledge-compilation formats and identify d-DNNFs as a sweet spot regarding capabilities for configuration counting and effort required to compile into the respective format. To expand the applicability of d-DNNFs for configuration counting, we provide optimized strategies to query a d-DNNF and two novel strategies to compile dependencies between features into a d-DNNF. Each of our proposals substantially outperforms the previous state of the art, scaling for vastly more systems and reducing runtimes from days to seconds in some cases. We already applied the proposals within industry collaborations enabling analysis of systems within seconds which did not scale at all with the previous state of the art.

Acknowledgements

Throughout my PhD studies, I was lucky to work with numerous amazing colleagues that were always eager to discuss ideas and support my endeavors. I am thankful for the always supportive and very enjoyable work environments with the teams at Ulm and Braunschweig. In particular, I highly appreciate the remarkable efforts by Matthias and Thomas to make our lives vastly easier when most of our group already moved away from Ulm and after the birth of my daughter Zoe.

Without the continuous support of all my colleagues, this thesis would not have been possible. Foremost, I want to thank my supervisor Thomas for luring me to research, always providing guidance, and supporting me in approaching problems my way. His continuous eagerness to listen to our issues and help to resolve them through my PhD journey has been astounding. I am grateful for having Paul as my partner in crime. Both of us were recruited by Thomas in the same period and in the end we submitted and defended our respective theses on the same days. He always found time for advise, sharing experiences in stressful times, and welcome distraction. I was also lucky to work with many outstanding students throughout my PhD studies. Many of my research contributions are heavily influenced by work within supervised theses. I want to particularly thank Heiko and Jan for their exceptional work as student assistants. In general, I am thankful for all the fruitful discussions and support by all of my colleagues. I could not have wished for more.

During my PhD studies, my co-authors were always eager to discuss ideas, provide feedback, and support me. Consequentially, I learned a lot from every one of my co-authors and they heavily influenced the contributions of this thesis. It is remarkable that every collaboration has been fruitful, inspiring, and also very pleasant on a personal level. I also want to thank everyone who provided feedback directly on my thesis, namely Thomas, the reviewers of my PhD thesis, all committee members, Sebastian, Paul, and Rahel. Their perspective had large impacts on the thesis itself. Furthermore, I am very thankful for all committed members for their interesting, thought-provoking questions during my PhD defense which inspire future research.

The support of my family and friends made this thesis and my life in general much easier and I am very thankful for that. A very special thanks to my wife Rahel, who has gone out of her way to support me by doing the lion's share of childcare when needed, lifting spirits and encouraging me in stressful periods, and always offering advice and support. I am also more than grateful for my sister Ronja who would always jump the fence to support me. Large thanks also to my mother Jutta who supported us a lot in general and particularly in the last weeks of this thesis. On that matter, I also want to thank my daughter Zoe for drastically improving her sleep schedule few weeks before the submission of this thesis.

List of Core Publications

The main contributions of this thesis are covered in the publications listed below. We discuss the relation to this thesis of (1) other research contributions during the author's PhD studies in Appendix A, (2) student's theses we advised in Appendix B, and (3) the student's thesis written by the author in Appendix C. The original manuscripts for each of the core publications can be found at the end of the thesis.

[A1] Chico Sundermann, Michael Nieke, Paul Maximilian Bittner, Tobias Heß, Thomas Thüm, and Ina Schaefer. "Applications of #SAT Solvers on Feature Models". In: *Proc. Int'l Working Conf. on Variability Modelling of Software-Intensive Systems (VaMoS)*. ACM, 2021. DOI: 10.1145/3442391.3442404

© 2021 Copyright held by the owner/author(s)

[A2C1] Chico Sundermann, Elias Kuiter, Tobias Heß, Heiko Raab, Sebastian Krieter, and Thomas Thüm. "On the Benefits of Knowledge Compilation for Feature-Model Analyses". In: *Annals of Mathematics and Artificial Intelligence (AMAI)* 92.5 (2024), pp. 1013–1050. DOI: 10.1007/s10472-023-09906-6

Published under Creative Commons Attribution 4.0 International (CC-BY), see https://creativecommons.org/licenses/by/4.0/.

[B1] Chico Sundermann, Tobias Heß, Michael Nieke, Paul Maximilian Bittner, Jeffrey M. Young, Thomas Thüm, and Ina Schaefer. "Evaluating State-of-the-Art #SAT Solvers on Industrial Configuration Spaces". In: *Empirical Software Engineering (EMSE)* 28.2 (2023), p. 38. DOI: 10.1007/s10664-022-10265-9

Published under Creative Commons Attribution 4.0 International (CC-BY), see https://creativecommons.org/licenses/by/4.0/.

[B2] Chico Sundermann, Thomas Thüm, and Ina Schaefer. "Evaluating #SAT Solvers on Industrial Feature Models". In: *Proc. Int'l Working Conf. on Variability Modelling of Software-Intensive Systems (VaMoS)*. ACM, 2020. DOI: 10.1145/3377024.3377025

© 2020 Copyright held by the owner/author(s)

[B3] Chico Sundermann, Vincenzo Francesco Brancaccio, Elias Kuiter, Sebastian Krieter, Tobias Heß, and Thomas Thüm. "Collecting Feature Models from the Literature: A Comprehensive Dataset for Benchmarking". In: *Proc. Int'l Systems and Software Product Line Conf. (SPLC)*. ACM, 2024, pp. 54–65. DOI: 10.1145/3646548.3672590

Published under Creative Commons Attribution 4.0 International (CC-BY), see https://creativecommons.org/licenses/by/4.0/.

[C2] Chico Sundermann, Heiko Raab, Tobias Heß, Thomas Thüm, and Ina Schaefer. "Reusing d-DNNFs for Efficient Feature-Model Counting". In: *Trans. on Software Engineering and Methodology (TOSEM)* 33.8 (2024). DOI: 10.1145/3680465

Published under Creative Commons Attribution 4.0 International (CC-BY), see https://creativecommons.org/licenses/by/4.0/.

[C3] Chico Sundermann, Jacob Loth, and Thomas Thüm. "Efficient Slicing of Feature Models via Projected d-DNNF Compilation". In: *Proc. Int'l Conf. on Automated Software Engineering (ASE)*. ACM, 2024, pp. 1332–1344. DOI: 10.1145/3691620.3695594

Published under Creative Commons Attribution 4.0 International (CC-BY), see https://creativecommons.org/licenses/by/4.0/.

[C4] Chico Sundermann, Stefan Vill, Elias Kuiter, Sebastian Krieter, Thüm, and Matthias Tichy. *Pseudo-Boolean d-DNNF Compilation for Expressive Feature Modeling Constructs*. Tech. rep. arXiv:2505.05976. Submitted to TSE. Cornell University Library, 2025. DOI: 10.48550/arXiv.2505.05976

Preprint published under CC-BY (https://creativecommons.org/licenses/by/4.0/), under review and will be published under same license if accepted.

Contents

Abstract iii

List of Core Publications vii

List of Figures xi

List of Tables xiii

1 *Introduction* 1

2 *Background* 7
- 2.1 Product Lines . . . 7
- 2.2 Logic-Based Reasoning . . . 9

3 *Application Scenarios for Configuration Counting* 15
- 3.1 Research Gap . . . 16
- 3.2 Collection of Applications . . . 17
- 3.3 Problem Settings in Configuration Counting . . . 21
- 3.4 Computational Effort of Applications . . . 22
- 3.5 Summary . . . 24

4 *Evaluating the State of the Art on Configuration Counting* 25
- 4.1 Research Gap . . . 26
- 4.2 Evaluate Performance . . . 27
- 4.3 Summary . . . 32

5 *Accelerating Configuration Counting with Knowledge Compilation* 33
- 5.1 Research Gap . . . 34
- 5.2 Knowledge Compilation for Feature-Model Analysis . . . 35
- 5.3 Repetitive Queries with d-DNNFs . . . 40
- 5.4 Projected d-DNNF Compilation . . . 44
- 5.5 Pseudo-Boolean d-DNNF Compilation . . . 49
- 5.6 Summary . . . 55

6 *Conclusion & Future Work* 57

Appendix 63
- A Related Work of the Author . . . 63
- B Advised Theses . . . 67
- C Student's Theses of the Author . . . 68

Bibliography 71

Original Core Publications 91

Chapter 3:
What applications are facilitated by which counting operations?

Chapter 4:
How well does the state of the art scale for configuration counting?

Chapter 5:
How can we scale to the numerous different computations required for configuration counting?

List of Figures

2 Running Example: Feature Diagram of Robot Vacuum . . . 9
11A Example Conjunctive Normal Form (CNF) . . . 11
12A Exhaustive DPLL Traversal for $(A \vee B) \wedge (\bar{A} \vee \bar{B} \vee C) \wedge (C \vee \bar{D})$ 12
14A Examples for d-DNNF Properties . . . 14
14B d-DNNF Created From Trace Figure *12A* . . . 14

18B Running Example: Feature Diagram of Robot Vacuum . . . 20

29A Runtime in Seconds for All Exact #SAT Solvers . . . 29
31A Runtime Automotive Histories . . . 31
31B Structural Properties Automotive Histories . . . 31

38A Methodology for Detecting Target Languages and Knowledge Compilers . . . 38
40A Runtimes Knowledge Compilers . . . 40
42A Computing Cardinalities on a d-DNNF . . . 42
44A Runtime Comparisons for Computing Cardinality of Features and Cardinality of Partial Configurations . . . 44
46A Commuting Diagram Showcasing Projected d-DNNF Compilation . . . 46
46B Projecting c and d in Exhaustive DPLL Traversal shown in Figure *12A* . . . 46
48A Performance Projected d-DNNF Compilation . . . 48
52A Pseudo-Boolean d-DNNF Compilation . . . 52

List of Tables

2A List of Main Contributions 2

10A Example Translation: Feature Diagram to Propositional Logic 10

18A Applications of Configuration Counting 18
22A Reducing Configuration Counting to Model Counting . . . 22
23A Number of Queries Required for Different Analyses 23

35A Mapping Between Core Contributions and Tackled Applications 35
38A Tractability of Feature-Model Analyses for Knowledge-Compilation Target Languages 38
54A Performance Comparison for CNF-Based and Pseudo-Boolean Pipeline 54

1 Introduction

Configurable systems are part of our everyday lives [89]. Most areas of life, such as transportation [48, 68, 129], telecommunication [37, 122], and computers [108, 220], offer various configuration options for their users. While tailored products offer many benefits, they also introduce a considerable complexity that needs to be handled [1, 69, 129, 211, 220].

Product lines are commonly employed to manage configurable systems. Each product line describes a family of products that share reusable units or features [89]. A single product can be composed by combining different features [89] instead of developing each product from scratch. Typically, not every combination or configuration of features leads to intended products. For instance, the inclusion of a feature may require including other features. We refer to dependencies and constraints that limit the valid combinations of features as *configuration knowledge*.

Managing configuration knowledge is essential to develop and maintain product lines. Major aspects of product-line engineering, such as configuration, testing, and development, rely on incorporating constraints between features. Otherwise, product developers may have to deal with unintended and possibly faulty products, and analyses on the system produce misleading results. Manual inspection of configuration knowledge in practice is infeasible since industrial product lines often contain thousands of features and constraints [B3]. Consequentially, a plethora of automated analyses that support understanding the configurability have been proposed [24, 147, 192, A1].

Various analyses depend on *configuration counting* (i.e., computing the number of valid configurations induced by a system) [76, 97, 129, 161, A2C1, A1]. Configuration counting can be beneficial through the entire product-line engineering process, in different application domains, and for various stakeholders [47, 100, 129, A1]. For instance, the number of valid configurations can provide an indication on the economic benefits of a product-line approach [47] or give users insights on the impact of a selection during configuration [146].

Research Gap & Contributions

Even though configuration counting has been considered in research and practice [76, 97, 129, 161, A2C1, A1], its applicability is limited by three main hurdles we identified. First, knowledge on applications facilitated by configuration counting is limited regarding their range of use and how to compute results [44, 76, 97, 129]. Second, the scalability of applying configuration counting in practice is largely unknown [76, 129, 136]. Third, as revealed by this thesis, scalable solutions for several problem settings in configuration counting are missing.

Main Objective of this Thesis

How to improve the applicability of configuration counting?

In this thesis, we aim to improve the applicability of configuration counting by gathering and extending potential use cases, and tackling its scalability. Table 2A provides an overview of our contributions. We target three main research objectives corresponding to the identified hurdles that explore (1) applications and problem settings of configuration counting, (2) the scalability of the state of the art, and (3) possible solutions to tackle the scalability for the identified problem settings. Each objective is motivated by research gaps and demands in practice. In particular, most of our contributions have been applied within our industrial collaboration with a large automotive company. With our proposals, we successfully tackled issues regarding scalability and understanding the variability of different product-line projects. In the following, we motivate and present the different contributions grouped by the research objective we are targeting.

Table 2A: *List of Main Contributions*

Research Objective	Ch.	Focus	Publication
Applications & Problem Settings	3	Collection of analyses	VaMoS'21 [A1]
		Computation instructions, complexity	AMAI'24 [A2C1]
Scalability State of the Art	4	General scalability #SAT	EMSE'23 [B1]
		Scalability #SAT during evolution	VaMoS'20 [B2]
		Collection of subject systems	SPLC'24 [B3]
Accelerate the Multitude of Queries	5	Knowledge compilation candidates	AMAI'24 [A2C1]
		Repetitive queries with d-DNNFs	TOSEM'24 [C2]
		Projected d-DNNF compilation	ASE'24 [C3]
		Pseudo-boolean d-DNNF compilation	Tech. Report [C4]
			Submitted to TSE

[A1] Sundermann et al. (2021), Applications of #SAT Solvers on Feature Models.
[A2C1] Sundermann et al. (2024), On the Benefits of Knowledge Compilation for Feature-Model Analyses.
[B1] Sundermann et al. (2023), Evaluating State-of-the-Art #SAT Solvers on Industrial Configuration Spaces.
[B2] Sundermann et al. (2020), Evaluating #SAT Solvers on Industrial Feature Models.
[B3] Sundermann et al. (2024), Collecting Feature Models from the Literature: A Comprehensive Dataset for Benchmarking.
[C2] Sundermann et al. (2024), Reusing d-DNNFs for Efficient Feature-Model Counting.
[C3] Sundermann et al. (2024), Efficient Slicing of Feature Models via Projected d-DNNF Compilation.
[C4] Sundermann et al. (2025), Pseudo-Boolean d-DNNF Compilation for Expressive Feature Modeling Constructs.

Research Objective 1 (Applications & Problem Settings)

What applications are facilitated by which counting operations?

While various applications for configuration counting have been considered in research [24, 47, 76, 97, 98, 161] and practice [96, 129], insights on potential use-case scenarios are limited and tedious to collect. In particular, the applications are (1) scattered over many publications [24, 47, 76, 98, 129, 161], (2) typically limited to specific application domains [47, 76, 98, 129, 161], and (3) often do not include a clear description on how to compute results [47]. As first major contribution, we provide a collection of applications based on configuration counting [A1]. The collection consists of applications we gathered from the literature and new proposals that became apparent within our industry collaborations. We further categorize the applications regarding underlying counting operations and the computational complexity [A2C1]. We observed that many applications rely on numerous counting operations, each of which are computationally expensive [31, 136, 218], indicating the need for scalable solutions.

Research Objective 2 (Scalability State of the Art)

How well does the state of the art scale for configuration counting?

Even though there are numerous applications dependent on configuration counting [24, 47, 76, 98, 129, 161], the scalability in practice was mostly unknown. Existing evaluations consider very few solvers [129, 136, 161], compare solving techniques not tailored for counting [174], or are limited to a single domain of subject systems [129]. Common analyses for understanding configurability typically operate on feature models [2, 16, 24, 28, 136, A1], which is a common specification mechanism for configuration knowledge [28, 114]. The de facto standard for feature-model analyses is translating the feature model to Boolean logic and use off-the-shelf solvers [16, 24, 52, 136, A2C1]. Model counting (#SAT for short) can be used to compute the number of satisfying assignments of a propositional formula [78, 182, 215] and, thus, the number of valid configurations for the previously translated feature model [97, 129, 136]. We aim to provide insights on the performance of #SAT technology for configuration counting. Overall, with a literature survey we gather and then evaluate 19 exact and 2 approximate #SAT solvers on industrial product lines, which cover various applications domains and a wide range of structural properties. Furthermore, we analyze the performance of #SAT solvers throughout the evolution of a system over time. One major observation is that #SAT scales to most industrial systems for single counting operations but applications requiring many queries may still exceed acceptable runtimes.

Research Objective 3 (Accelerate the Multitude of Queries)

How can we scale to the numerous different computations required for configuration counting?

Following the insights on possible applications and scalability of available solutions, we identified different problem settings that may hinder scalability of configuration counting and, thus, the adoption of it in practice. One main factor limiting the scalability of configuration counting is that numerous similar counting operations are required for most applications. As even counting once is computationally demanding [31, 78, B1, 218], the multitude of operations may exceed acceptable response times even for less complex systems. Our overall strategy to tackle the scalability for configuration counting relies on *knowledge compilation*. With knowledge compilation, the feature model is translated with an initial effort to a target language with beneficial properties for efficiently computing subsequent operations (typically called queries) [56]. A popular example for a target language are ordered binary decision diagrams (OBDDs) which support, amongst others, poly-time queries for SAT and #SAT [39, 99]. With four core contributions that revolve around knowledge compilation, we provide scalable solutions for several problem settings that cover the vast majority of configuration-counting applications. As our first contribution, we examine knowledge compilation target languages and derive d-DNNFs as sweet spot for configuration counting regarding compilation efforts and enabled counting operations [A2C1]. With our second contribution, we realize optimized algorithms to reuse d-DNNFs which enable efficient counting queries [C2].

As these contributions clearly indicate the benefits of reusing d-DNNFs, we propose two novel compilation techniques [C3, C4] to enable d-DNNFs for more application scenarios. In some scenarios, only a part of the feature model is relevant to a certain stakeholder [2, 94, 96, 126]. For instance, a software engineer may conduct analyses specific to software features.

However, acquiring such a projection (e.g., with only software features) is computationally expensive with available techniques, such as feature-model slicing [2, 126]. As third contribution targeting the third research objective, we propose *projected d-DNNF compilation* [C3], which merges two complex operations, namely projection [9, 56]/slicing [2, 126] and regular d-DNNF compilation [54, 133, 152], into one step. Our approach can also help to provide some insights on complex feature models where we cannot employ configuration counting for all features [C3]. Also, the applicability of available solutions is mostly limited to conjunctive normal form (CNF) as input which is ill-suited for several variability-modelling constructs [129, 136, 161, 193]. With *pseudo-Boolean d-DNNF compilation*, our fourth contribution tackling scalability issues of configuration counting, we target the mismatch between such constructs and the availability of scalable reasoning engines.

Overall, we make the following main contributions in this thesis.

- **Research Objective 1 (Applications & Problem Settings)**
  - We gather a collection of applications based on configuration counting that showcases its significance (**VaMoS'21** [A1]).
  - We classify each application regarding the underlying computational problem and the number of counting queries that are required (**AMAI'24** [A2C1]).
- **Research Objective 2 (Scalability State of the Art)**
  - We conduct a large-scale empirical evaluation on 19 exact and 2 approximate #SAT solvers to examine the performance of the state of the art for configuration counting (**EMSE'23** [B1]).
  - We analyze the performance of current counting strategies during the evolution of a system (**VaMoS'20** [B2]).
  - We gather a collection of 2,518 real-world feature models with a systematic literature survey to facilitate future empirical research (**SPLC'24** [B3]).
- **Research Objective 3 (Accelerate the Multitude of Queries)**
  - We perform a literature survey to collect available knowledge compilation target languages and examine their potential for configuration counting (**AMAI'24** [A2C1]).
  - We provide optimized algorithms to perform counting queries on d-DNNFs and showcase its benefits on industrial feature models (**TOSEM'24** [C2]).
  - We propose a novel strategy to derive d-DNNFs that need a projection of a feature model with projected d-DNNF compilation (**ASE'24** [C3]).
  - We expand the applicability of d-DNNF compilation for expressive feature-modeling constructs that are ill-suited for propositional reasoning with pseudo-Boolean d-DNNF compilation (**TSE Subm.** [C4]).

Thesis Structure

The thesis is organized around the three main topics outlined in Table 2A. After providing the necessary background in Chapter 2 to understand

the concepts used in this thesis, we present major insights from the listed publications and discuss our contributions for configuration counting in the bigger picture. For each major topic, we examine relevance, review the research gap in related work, present our contributions, and discuss potential future work. While we provide an overview for each contribution, we refer to the original publications for more details. In Chapter 3, we present the identified application scenarios. In Chapter 4, we discuss our empirical results on performance of state-of-the-art solutions for configuration counting. In Chapter 5, for each identified research gap, we provide solutions that accelerate configuration counting compared to the state of the art. Finally, we discuss the overall advancements of our thesis in research on configuration counting and reflect on potential future research in Chapter 6.

2 Background

1. Product Lines 7
2. Logic-Based Reasoning 9

In this chapter, we provide the required background to understand the following chapters and introduce several conventions we use throughout the thesis. We cover relevant concepts in the context of product lines (Section 2.1) and automated reasoning based on logic (Section 2.2).

2.1 Product Lines

A *product line* describes a family of related products that share reusable *features* [115]. Individual products are not developed from scratch but can be composed by combining different features [115]. Product lines are prevalent in various areas of life, such as configuring vehicles [96, 129, 163, 201, 226], operating systems on different personal devices [81, 132, 145], all types of end-user software [88, 149, 189, B3], or offering customized loans [79]. Typically, not every combination of features in such product lines leads to a desired product [115, 142]. For instance, some features may rely on other features to operate as expected.

2.1.1 Configuration Knowledge

The *configuration knowledge* specifies which combinations of features are considered as desired products of the product line [74]. Various representations of configuration knowledge have been considered in the literature [21, 28, 74, 114], such as natural language [28], product enumeration [28], logical formulas [28, 29, 171], or variability models [21, 62, 114, 173]. A *variability model* is a collective term for specifications dedicated to describing the commonalities and variabilities of a product line [28, 29].

2.1.2 Feature Models

Feature models are a commonly used type of variability models [3, 16, 28, 84, 114, 148, 149]. In a survey of Berger et al. [28] about industrial habits in product-line engineering, feature models are the most popular specification for configuration knowledge. Feature models specify the *valid configurations* of a product line by denoting a set of features and constraints over those features [114]. Depending on the feature-modeling dialect, constraints can have various structures, such as logical formulas over the feature set or different kinds of hierarchical parent-child relationships [21, 25, 28]. In this thesis, most concepts do not rely on a specific structure of a constraint. Hence, if not stated otherwise, we consider constraints rather as a function that specifies whether that constraint is consistent or conflicting with a given *configuration*. A configuration, that is consistent with every constraint $c \in \mathbb{CT}$ imposed by the feature model, is called *valid*.

Definition 1 (Feature Model)**.** A *feature model* $\mathbb{FM} = (\mathbb{FS}, \mathbb{CT})$ is defined by a set of features $\mathbb{FS}$ and a set of constraints $\mathbb{CT}$ that limit the set of valid configurations for $\mathbb{FM}$.

Definition 2 (Configuration)**.** A *configuration* $\mathbb{C}_{\mathbb{FM}} = (\mathbb{I}, \mathbb{E})$ for $\mathbb{FM} = (\mathbb{FS}, \mathbb{CT})$ is a tuple of two feature sets denoting the included features $\mathbb{I} \subseteq \mathbb{FS}$ and the excluded features $\mathbb{E} \subseteq \mathbb{FS}$. In a configuration, no feature is included and excluded simultaneously (i.e., $\mathbb{I} \cap \mathbb{E} = \varnothing$). We refer to a configuration in which every feature is either included or excluded (i.e., $\mathbb{I} \cup \mathbb{E} = \mathbb{FS}$) as *complete*. Otherwise, we call it *partial*. $\mathbb{C}^*_{\mathbb{FS}}$ refers to the set of all possible complete configurations over a feature set $\mathbb{FS}$ (i.e., $\mathbb{C}^*_{\mathbb{FS}} = \{(\mathbb{I}, \mathbb{E}) | \mathbb{I} \cup \mathbb{E} = \mathbb{FS}, \mathbb{I} \cap \mathbb{E} = \varnothing\}$).

Definition 3 (Constraint)**.** A *constraint* $c \in \mathbb{CT} : \mathbb{C}^*_{\mathbb{FS}} \to \{\top, \bot\}$ is a function that denotes whether a given configuration $\mathbb{C}$ is consistent ($c(\mathbb{C}) = \top$) or contradicting ($c(\mathbb{C}) = \bot$) with the constraint c.

Definition 4 (Valid Configuration)**.** A complete configuration $\mathbb{C}_{\mathbb{FM}} \in \mathbb{C}^*_{\mathbb{FS}}$ for a feature model $\mathbb{FM} = (\mathbb{FS}, \mathbb{CT})$ is *valid* if the configuration is consistent with every constraint (i.e., $\forall c \in \mathbb{CT} : c(\mathbb{C}_{\mathbb{FM}}) = \top$).

Definition 5 (Configuration Space)**.** The *configuration space* $\mathbb{CS}_{\mathbb{FM}} \subseteq \mathbb{C}^*_{\mathbb{FS}}$ is the set of all valid complete configurations for a feature model $\mathbb{FM} = (\mathbb{FS}, \mathbb{CT})$ (i.e., $\mathbb{CS}_{\mathbb{FM}} = \{(\mathbb{I}, \mathbb{E}) \,|\, (\mathbb{I}, \mathbb{E}) \in \mathbb{C}^*_{\mathbb{FS}}, \forall c \in \mathbb{CT} : c((\mathbb{I}, \mathbb{E})) = \top\}$).

Feature diagrams are a common visualization of feature models [52, 148, 187] and are used throughout this thesis. Figure 2 displays a feature diagram specifying the configuration space of a robot vacuum product line, which we use as a running example throughout the thesis. The constraints $\mathbb{CT}$ in the feature diagram consist of hierarchical parent-child relationships and propositional cross-tree constraints (CTC for short). In our example, each `robot vacuum` requires `software` and `hardware` as denoted by their *mandatory* flag. The software may include `maps`, `obstacle detection`, and a `mop mode`, while the hardware may include a `camera` and a `storage`, as denoted by their *optional* flags. Furthermore, one or both of `visual` and `physical` must be selected after including `obstacle detection` as indicated by the *or* group. The *alternative* group of `storage` indicates that exactly one storage component can be included. Selecting a feature always mandates to select its parent feature. In addition to the hierarchical groups, there are cross-tree constraints which can be arbitrary propositional formulas over the set of features. In our running example, the cross-tree constraints specify that selecting `visual` or `maps` requires a `camera` and `mop mode` depends on `water storage`.

2.1.3 Feature-Model Analysis

In practice, feature models commonly contain thousands of features and constraints [29, 118, 129, 145, B3]. Hence, manual inspection of the configuration knowledge is generally infeasible [24]. A plethora of automated analyses have been proposed [24, 91, A2C1, A1]. Those analyses consider tasks from various stages of product-line engineering, such as management [47, 97], development [24], testing [91], modeling [24, 188], and configuring [128]. Available analyses often rely on common operations [A2C1], such as checking whether a given configuration is valid [24, 128], whether

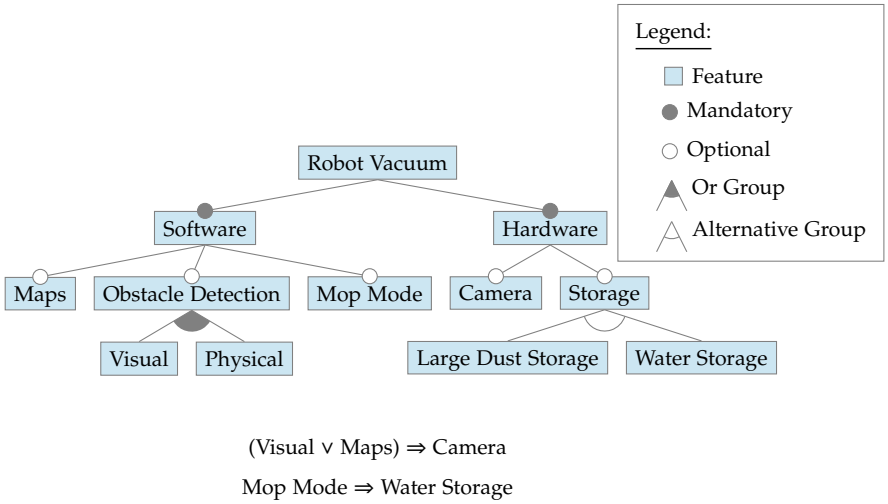


Figure 2: *Running Example: Feature Diagram of Robot Vacuum*

the feature model induces any valid configuration at all [24], generating a set of sample configurations following an objective [112, 160, 172], or computing the number of valid configurations [24, 47, 96, 97, 129, 161, A1].

We introduce *configuration counting* as a collective term over operations that compute the number of valid configurations in different settings. Formally, we consider the number of all valid configuration induced by a feature model $\mathbb{FM}$ as cardinality of its configuration space (i.e., $|\mathbb{CS}_{\mathbb{FM}}|$). Configuration counting facilitates a variety of automated analyses [24, 47, 97, 129, 161] which have been considered in various application domains [47, 96, 129, 161] and for plenty of different tasks in product-line engineering [24, 47, 97, 162]. In our work on applications of configuration counting [A2C1, A1], we gathered different applications that rely on analyzing different subsets of the configuration space. We discuss example applications and the relevant configuration-space subsets in Chapter 3.

2.2 Logic-Based Reasoning

The de facto standard for analyzing feature models is translating them to a logical expression (typically propositional) and invoking off-the-shelf solvers [16, 24, 129, 136, 161]. Here, the different operations typical analyses rely on are reduced to logic problems [97, 136, A2C1]. For instance, checking whether a feature model induces any valid configuration can be reduced to SAT [136], while configuration counting can be reduced to #SAT [A1]. Other more complex analyses, such as t-wise sampling [127] and decision propagation during the configuration process [128], are also often reduced to multiple solver invocations [A2C1].

2.2.1 Propositional Representations

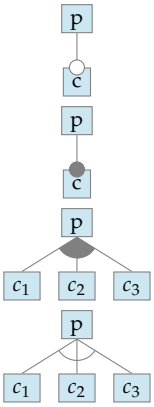
Each *propositional formula* F connects *literals*, which correspond to the underlying set of variables $V(F)$, with *logical operators* [109]. Common operators are $\wedge$ (conjunction), $\vee$ (disjunction), $\neg$ (negation), $\Rightarrow$ (implies), and $\Leftrightarrow$ (iff). A formula F can evaluate to $\top$ (true) or $\bot$ (false) for a given *assignment*

α, which assigns variables $v \in V(F)$ to $\top$ or $\bot$. Assignments behave analogously to configurations by specifying two sets that describe the variables set to $\top$ (included features for configurations) or to $\bot$ (excluded features).

Definition 6 (Assignment)**.** An *assignment* $\alpha = (\checkmark, \times)$ is a tuple of two sets that contain the variables set to $\top$ ($\checkmark$) and variables set to $\bot$ ($\times$), respectively. For brevity, we use $\alpha(F)$ to denote whether an assignment α satisfies (i.e., $\alpha(F) = \top$) the formula F or not (i.e., $\alpha(F) = \bot$). An assignment α is *complete* if every variable $v \in V(F)$ is assigned (i.e., $\checkmark \cup \times = V(F)$). Otherwise, the assignment α is *partial*. We use $A(F)$ to denote the set of all satisfying complete assignments for F (i.e., $A(F) = \{\alpha \mid \alpha(F) = \top, \checkmark \cup \times = V(F), \checkmark \cap \times = \varnothing\}$).

Feature models are typically prepared for automated analysis by translating them to propositional logic with the following procedure [67, 136, A2C1]. For a given feature model $\mathbb{FM} = (\mathbb{FS}, \mathbb{CT})$, the constraints $\mathbb{CT}$ over features $\mathbb{FS}$ are translated to a formula $F_{\mathbb{FM}}$ over variables $V(F_{\mathbb{FM}})$ with $V(F_{\mathbb{FM}}) = \mathbb{FS}$. The concrete transformation to a logical representation depends on the types of constraints $\mathbb{CT}$ and the logical dialect. As an example, we show the transformation of feature diagrams (cf. Figure 2) to propositional logic, which are both commonly employed in feature-model analysis [52, 97, 136, 148]. Here, we translate each parent-child relationship, namely *optional*, *mandatory*, *or*, *alternative*, to semantically equivalent propositional expressions. The cross-tree constraints of feature diagrams are typically already in propositional logic [84, 148, 203]. We consider the resulting formula $F_{\mathbb{FM}}$ as *semantically equivalent* to $\mathbb{FM}$ if every valid configuration $(\mathbb{I}, \mathbb{E}) \in \mathbb{CS}_{\mathbb{FM}}$ is a satisfying assignment for the formula $F_{\mathbb{FM}}$ (i.e., $(\mathbb{I}, \mathbb{E}) \in A(F_{\mathbb{FM}})$) and vice versa.

Table 10A: *Example Translation: Feature Diagram to Propositional Logic*

Feature-Modeling Construct	Propositional Representation
	$c \Rightarrow p$ $c \Leftrightarrow p$ $p \Leftrightarrow (c_1 \vee c_2 \vee c_3)$ $(p \Leftrightarrow (c_1 \vee c_2 \vee c_3)) \wedge (\bar{c_1} \vee \bar{c_2})$ $\wedge(\bar{c_2} \vee \bar{c_3}) \wedge (\bar{c_1} \vee \bar{c_3})$
Cross-tree constraint ϕ	ϕ

Definition 7 (Equivalence of Feature Model and Formula)**.** A feature model $\mathbb{FM} = (\mathbb{FS}, \mathbb{CT})$ and a formula F are semantically equivalent iff both are defined over the same variables (i.e., $\mathbb{FS} = V(F)$) and the set of valid configuration is equal to the set of satisfying assignments (i.e., $\mathbb{CS}_{\mathbb{FM}} = A(F)$).

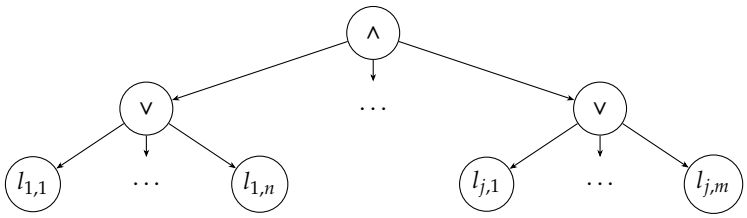


Figure 11A: *Example Conjunctive Normal Form (CNF)*

To use off-the-shelf solvers, the propositional formulas are typically transformed to a normal form [30, 78, 109, 215]. The de facto standard for Boolean solvers is the *conjunctive normal form* (CNF) [30, 182, 215]. Figure 11A visualizes the structure of a CNF $(l_{1,1} \vee \ldots \vee l_{1,n}) \wedge \ldots \wedge (l_{j,1} \vee \ldots \vee l_{j,m})$ using a *directed acyclic graph*. A CNF is a single conjunction of one or more *clauses* [109]. Each clause is a single disjunction of at least one literal. Note that every CNF is also in *negation normal form* (i.e., only literals can be negated). Every propositional formula can be represented as CNF [175]. Furthermore, there are CNF transformations (e.g., Tseitin [175]) where only a linear number of clauses w.r.t. the size of the original formula are generated [131, 175]. If not stated otherwise, we consider each propositional formula to be in CNF.

Definition 8 (Negation Normal Form [109])**.** A formula F is in *negation normal form (NNF)* if the operators are limited to $\vee, \wedge, \neg$ (and parentheses), and $\neg$ is only applied directly to literals. To denote a negated literal $\neg l$, we use $\bar{l}$ throughout this thesis.

Definition 9 (Conjunctive Normal Form [109])**.** A formula F is in *conjunctive normal form* (CNF) if F is a conjunction of disjunctions that only contain literals. Every CNF is an NNF. Formally, a CNF F always follows the structure $F = \bigwedge_{i=1}^{n}(\bigvee_{j=1}^{m} l_{i,j})$ with $n, m \in \mathbb{N}$ and $l_{i,j}$ being literals corresponding to variables $v \in V(F)$.

2.2.2 Model Counting

Model counting (#SAT for short) refers to computing the number of satisfying assignments for a given formula [31]. In practice, model counts are used for many different tasks in a variety of application domains [86, 195, 212]. For instance, model counting is used to examine the reliability of a program by computing the ratio of failed executions [212], to verify neural networks [15], for automated attacks on encrypted messages [20], or to *compute the number of valid configurations* [A1].

Definition 10 (Model Count)**.** The *model count* #SAT(F) refers to the number of satisfying assignments for a formula F (i.e., #SAT(F) = $|A(F)|$).

The #SAT problem is #P-complete [218]. Each problem in #P corresponds to a respective decision problem that is in NP [31]. It is obvious that the #SAT problem is at least as hard as SAT [31]. If the number of satisfying assignments is known, it is trivial to derive whether the formula is satisfiable (i.e., SAT(F) $\equiv$ (#SAT(F) ≥ 1)). While the basic approaches for both SAT (i.e., DPLL [137]) and #SAT (i.e., exhaustive DPLL [33]) have the same worst-case complexity, #SAT is generally considered to be harder to

solve [31]. Also, poly-time specializations of the SAT problem, such as 2-SAT or Horn-SAT, still have an exponential computational complexity for #SAT [31].

Current approaches for model counting typically rely on an exhaustive DPLL (short for Davis-Putnam-Logemann-Loveland) traversal of the solution space [33, 40, 182, 194, 215] or querying knowledge compilation artifacts [31, 55, 56, 78]. Exhaustive DPLL is typically performed on CNFs to simplify updating the formula during traversal [33, 54, 133, 182, 215]. Figure 12A shows the trace for an exhaustive DPLL traversal for the formula $(a \vee b) \wedge (\bar{a} \vee \bar{b} \vee c) \wedge (c \vee \bar{d})$. To explore the satisfying assignments, the algorithm branches (i.e., assigns variables) until a clause is falsified or each clause is satisfied. Then, the respective subformula is updated according to this assignment. For an assignment v, clauses containing the literal v are removed from this branch as they are satisfied and literals $\bar{v}$ are removed from clauses as this literal can no longer be satisfied. For instance in Figure 12A, assigning a satisfies the first clause $(a \vee b)$ and falsifies $\bar{a}$ in $(\bar{a} \vee \bar{b} \vee c)$ which overall results in the formula conditioned subformula $(\bar{b} \vee c) \wedge (c \vee \bar{d})$. When a clause is falsified or each clause is satisfied, the algorithm backtracks to find the next satisfying or falsifying assignment. This procedure is repeated until all branches have been found either satisfied or falsified. In each step, we assign a variable (starting with a here) and then update the clauses according to the partial assignment in this branch. If each clause is satisfied, the number of satisfying assignments in this branch is 2^n where n is the number of open (i.e., not assigned to $\top$ or $\bot$) variables. For a falsified branch, the number of satisfying assignments is zero. After full traversal, the sum of the different branches is the model count of the entire formula. In Figure 12A, we already see two common optimizations: *Boolean constraint propagation* and *disconnected components* [18]. When having a *unit clause* (i.e., a clause with exactly one literal) (v), the formula can only be satisfied under v. Hence, we can directly perform Boolean constraint propagation to set v. For instance, this is performed on $c \wedge (c \vee \bar{d})$ in the leftmost branch. When identifying disconnected components (i.e., sets of clauses that share no variables), these components can be traversed individually [18] as seen in our example Figure 12A for $b \wedge (c \vee \bar{d})$. For counting, we need to multiply the model counts of the respective components to get the overall count of this branch.

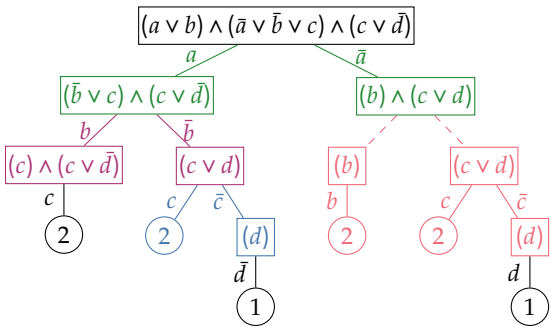


Figure 12A: *Exhaustive DPLL Traversal for* $(A \vee B) \wedge (\bar{A} \vee \bar{B} \vee C) \wedge (C \vee \bar{D})$

2.2.3 Knowledge Compilation

With *knowledge compilation*, the original formula (typically in CNF) is translated with an initial effort to a *target language* for which more operations

are *tractable* than for the original formula [56]. Typical operations considered for knowledge compilation are checking satisfiability, model counting, model enumeration, or algebraic operations (e.g., conjunction) [56]. The advantage of knowledge compilation is that after the initial computationally expensive compilation, the effort can be amortized by processing subsequent queries more efficiently. Hence, knowledge compilation is especially beneficial when many queries are demanded.

Definition 11 (Tractability [56])**.** A logical operation P is *tractable* for a language if queries of P are solvable on every expression of that language in poly-time with respect to the size of the expression.

Definition 12 (Knowledge Compilation Target Language [56])**.** A language L is considered as *knowledge compilation target language* if there are logical problems that are tractable for the language L but not for CNF.

There are plenty of knowledge compilation target languages that enable different poly-time queries [56, 120, 165]. When considering promising target languages, there is a tradeoff between computational effort and relevant tractable operations on that language. A popular language are *(ordered) binary decision diagrams*, which are hard to compile but enable various operations [56, 213]. Given a problem setting defined by the kinds of targeted operations, one may seek the target language that is the easiest to compile, but for which all targeted operations are still tractable [56]. A tool for compiling the original formula to a target language is called a *knowledge compiler* [78, 134, 165].

2.2.4 d-DNNF

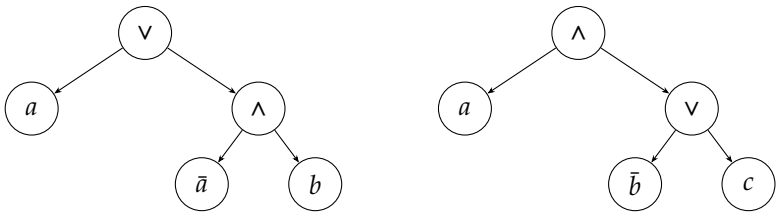
The *deterministic decomposable negation normal form (d-DNNF)* is a knowledge compilation target language that has attracted attention over the last years [36, 133, 152, 193]. Exploiting the properties determinism and decomposability enables linear-time queries (w.r.t. expression size) for checking satisfiability, model counting, and model enumeration on d-DNNFs [56].

Definition 13 (d-DNNF [54])**.** A formula is in *deterministic decomposable negation normal form (d-DNNF)* if (i) every disjunction is deterministic, (ii) every conjunction is decomposable, and (iii) the formula is in negation normal form.

Definition 14 (Determinism [54])**.** A disjunction $d_1 \vee \ldots \vee d_n$ is *deterministic* if its disjuncts d_i are logically disjunct (i.e., $d_i, d_j : i \neq j : d_i \wedge d_j \equiv \bot$).

Definition 15 (Decomposability [54])**.** A conjunction $c_1 \wedge \ldots \wedge c_n$ is *decomposable* if its conjuncts c_i share no variables (i.e., $c_i, c_j : i \neq j : V(c_i) \cap V(c_j) = \varnothing$).

The disjunction shown in Figure 12Aa is deterministic as every satisfying assignment on the left contains a (i.e., $\alpha(a) = \top$) while every satisfying assignment on the right contains $\bar{a}$ (i.e., $\alpha(a) = \bot$). The conjunction shown in Figure 12Ab is decomposable as the left side and right side share no variables (i.e., $\{a\} \cap \{b, c\} = \varnothing$).



(a) *Example Deterministic Disjunction* (b) *Example Decomposable Conjunction*

Figure 14A: *Examples for d-DNNF Properties*

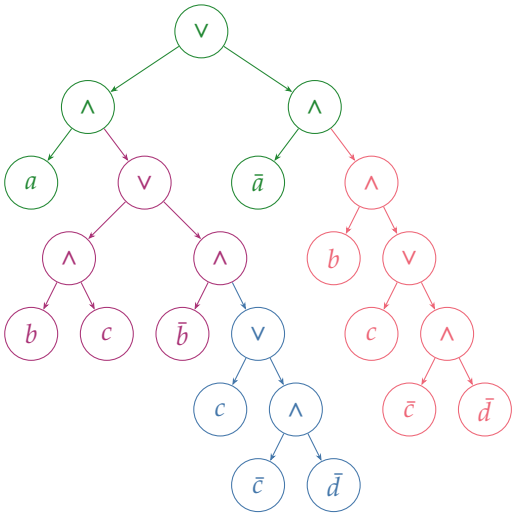


Figure 14B: *d-DNNF Created From Trace Figure 12A*

Current d-DNNF compilers derive the d-DNNF from the trace of an exhaustive DPLL trace [54, 133, 152]. Figure 14B shows the d-DNNF derived from the trace shown in Figure 12A. Each time we branch on a variable v for the current subformula ϕ, we create a disjunction $(\phi \wedge v) \vee (\phi \wedge \bar{v})$. Note that this disjunction is deterministic by construction. Every time we split the current subformula ϕ according to identified disconnected components ϕ_X and ϕ_Y, we create a conjunction $\phi_X \wedge \phi_Y$. As the components are disconnected, the variables $X = V(\phi_X)$ and $Y = V(\phi_Y)$ of the components share no variables (i.e., $X \cap Y = \varnothing$). Note that the conjunction $\phi_X \wedge \phi_Y$ is decomposable by construction. Following these instructions, we can traverse the #SAT trace (or already do it during the exhaustive DPLL traversal) to derive a d-DNNF that is semantically equivalent to the input formula [54, 133].

3 *Application Scenarios for Configuration Counting*

1. Research Gap 16
2. Collection of Applications 17
3. Problem Settings in Configuration Counting 21
4. Computational Effort of Applications 22
5. Summary 24

Research Objective 1 (Applications & Problem Settings)

Central Research Question

What applications are facilitated by which counting operations?

Core Publications

VaMoS'21 [A1]: "Applications of #SAT Solvers on Feature Models"
Chico Sundermann, Michael Nieke, Paul Maximilian Bittner, Tobias Heß, Thomas Thüm, and Ina Schaefer, *Proc. Int'l Working Conf. on Variability Modelling of Software-Intensive Systems (VaMoS)*, 2021, Manuscript in Appendix C.1

AMAI'24 [A2C1]: "On the Benefits of Knowledge Compilation for Feature-Model Analyses"
Chico Sundermann, Elias Kuiter, Tobias Heß, Heiko Raab, Sebastian Krieter, and Thomas Thüm, *Annals of Mathematics and Artificial Intelligence (AMAI)*, 2024, Manuscript in Appendix C.2

Over the last decade, benefits of configuration counting have been widely recognized. Applications relying on configuration counting have been considered in various settings [47, 97, 129, A1]. In particular, configuration counting may be relevant for different stakeholders in product-line engineering [47, 129, 146, 161] and in a variety of application domains [47, 96, 129, 161], such as automotive systems [96, 129]. Still, applying configuration counting is non-trivial since knowledge on existing applications and, thus, potential benefits is not easily accessible. In the literature, available applications are scattered across publications, instructions on how to compute results are often lacking, and computational effort of the analyses is commonly not discussed. In this chapter, we discuss the contributions of our thesis that provide insights on possible applications of configuration counting [A2C1, A1]. We provide a collection of applications based on configuration counting including computation instructions and discussions on the required computational effort.

This chapter is structured as follows. In Section 3.1, we discuss the identified research gap regarding the availability of knowledge on applications of configuration counting. Here, we discuss its relevance and the limitations of related work. We present our collection of applications that showcases the possibilities facilitated by configuration counting [A1] in Section 3.2. To simplify computing results for the applications, we provide a classification targeting the respective computational problem settings [A2C1, A1] in Section 3.3. In Section 3.4, we examine the computational effort induced by the different applications [A2C1].

3.1 Research Gap

In this section, we discuss the research gap we identified in available information on applications of configuration counting. We examine the practical relevance of configuration counting and our proposal of providing a unified collection. Furthermore, we discuss the shortcomings and limitations of related work on applications of configuration counting.

3.1.1 Relevance

Configuration counting can be beneficial for various aspects of product-line engineering. In particular, applications relying on configuration counting have been proposed for planning product-line projects [47], supporting development [129], test-case generation [161], configuration processes [44, 146], and statistical analysis [98]. The insights gained from configuration counting are also relevant for various stakeholders, such as users [146, A1], software developers [96], variability-modeling experts [129], and management [47]. For instance, management can use configuration counting for estimating economic benefits of the product line [47] and users can be supported in understanding the impact of a feature selection with the number of remaining valid configurations [44, 146].

Easily accessible information on possible applications of configuration counting is important for its adoption. As many discussions with our industry partners revealed, recognizing the option to apply existing analyses for problems occurring in practice is not trivial. Recognizing the opportunity to apply configuration counting is even harder when there is no convenient source of knowledge on the possible applications. To facilitate identifying and understanding potential application scenarios, an accessible collection of applications is helpful. The success[1] of related collections on product-line analysis [24, 147, 214] indicates the benefits of having such a collection for configuration counting. In the following section, we demonstrate the lack of such a collection and discuss why existing work makes it complex and tedious to identify and evaluate potential applications.

1. Here, we use citation count as indicator for success. Each of the listed surveys has been cited over 200 times according to GoogleScholar as of January 2025.

3.1.2 Limitations of Related Work

The applications considered in literature and practice indicate the benefits and capabilities of configuration counting. However, with existing work, it is *tedious to identify relevant applications* and get an overview on the possibilities opened up by configuration counting. While there are some collections targeting applications of configuration counting [76, 97, 129], these are limited to few applications (i.e., five or less). Furthermore, many applications are scattered over surveys that have a different main scope [11, 24, 36, 44, 98, 146, 168, 174], such as metrics describing complexity of a system [11] or general feature-model analyses [24]. Other publications consider configuration counting only in a specific application area [11, 35, 44, 47, 76, 97, 146, 167], such as economic estimations [35, 47, 76, 97, 98, 161, 193] or the configuration process [44, 146]. Overall, collecting the relevant applications requires researchers and practitioners to review numerous publications. Even worse, many applications are hidden in publications in which the consideration of configuration counting is not obvious.

Within industrial collaborations, we identified several applications of configuration counting. Most of these applications have not previously been

considered in the literature. These observations indicate that the possibilities of configuration counting have not been fully explored in literature and that further proposals are valuable.

Another frequent problem for using the applications in practice is the *lack of computation instructions*. In particular, several publications present applications based on knowing the number of valid configurations but provide no way to compute the number [47, 146]. Other publications discuss general ideas how to compute the number of valid configurations but no precise instructions for specific applications [24, 36, 97]. In many cases, if instructions are provided, they are based on solutions, with OBDDs being the most prevalent approach [11, 24, 98, 168, 174], that have been shown to not scale for many industrial systems [23, 103, 174]. Few publications [76, 129] provide clear and potentially scalable computation instructions, but they focus on specific problem settings that can only be applied for some other existing applications [76] or do not provide sufficient information to generalize the given instructions [129]. Consequently, even if a user finds applications relevant for their use case, it is often hard to find clear, unambiguous, and scalable computation instructions.

The computational complexity of applications based on configuration counting is typically not considered in the literature. While the complexity of a single computation (e.g., invocation of #SAT) is sometimes discussed [76, 97, 129], the *number of required computations is typically overlooked*. However, the number of required computations is an essential factor for considering the scalability, as there are large differences between applications. Some applications only require computing the number of valid configurations once, while others may induce thousands or even millions of #SAT invocations [A2C1], even for a single system version. Hence, it is difficult for researchers and practitioners to assess the scalability and consequentially the applicability for found applications.

3.2 Collection of Applications

To tackle the discussed shortcomings of related work, we provide an expert survey of applications depending on configuration counting. The collection consists of applications we found in the literature and new applications mostly stemming from discussions within industry collaborations. To describe the applications and instructions on how to compute results, we use feature models as formalism for specifying the valid configurations.

Table *18*A provides an overview on each application presented in our **VaMoS'21** publication [A1]. Overall, we gathered 21 applications consisting of 12 applications from literature and 9 applications inspired by our industrial collaborations. The applications target various inevitable tasks of product-line engineering, namely planning (e.g., payoff thresholds), development (e.g., feature prioritization), testing (e.g., rating errors), configuring (e.g., optimize configuration process), and statistics (e.g., homogeneity). Furthermore, insights from the applications may be relevant for involved stakeholders, such as modeling experts, developers, testers, managers, and users.

We classified each application by the part of the configuration space it targets. The goal of the classification is twofold. First, we aim to facilitate recognizing the potential of applying configuration counting and the identification of concrete applications for problems occurring in product-line

Table 18A: *Applications of Configuration Counting*

Application	Source	Year
Cardinality of feature model $\lvert\mathbb{CS}_{\mathbb{FM}}\rvert$		
Variability factor	Benavides et al. [23],	2005
	Heradio et al. [97, 100]	2011
	Fernandez et al. [76],	2014
Degree of orthogonality	Czarnecki et al. [51]	2005
	Benavides et al. [24]	2010
Cost savings of a product line	Clements et al. [47]	2005
	Heradio et al. [97]	2013
Void feature model	Benavides et al. [24]	2010
Maintainability prediction	Bagheri et al. [12]	2012
Variability reduction	**New**	2021
Configuration relevance	**New**	2021
Cardinality of feature $\lvert\mathbb{CS}_{f}\rvert$		
Degree of reuse	Cohen et al. [49, 76, 97]	2003
Homogeneity	Clements et al. [47],	2005
	Fernandez et al. [76],	2014
	Heradio et al. [97, 100]	2011
Core, dead & false-optional	Trinidad et al. [217]	2006
	Perez [168]	2016
Payoff threshold of a feature	Heradio et al. [97, 100]	2011
Optimize configuration process	Chen et al. [44]	2011
	Mazo et al. [146]	2014
Atomic set candidates	**New**	2021
Feature prioritization	**New**	2021
CTC restrictiveness for features	**New**	2021
Cardinality of partial configuration $\lvert\mathbb{CS}_{\mathbb{C}}\rvert$		
Uniform random sampling	Oh et al. [160, 161],	2017
	Achlioptas et al. [6],	2018
	Munoz et al. [153],	2019
	Sharma et al. [193],	2018
	Charkchakraborty et al. [42]	2015
Atomic sets	**New**	2021
Rating feature interactions	**New**	2021
Interactive configuration support	**New**	2021
Cardinality of formula $\lvert\mathbb{CS}_{\mathbb{FM}'}\rvert$		
Variability of feature subsets	**New** (used in [96])	2021
Rating errors	Kübler et al. [129]	2010

engineering. Second, depending on the part of the feature model that is of interest, different reasoning engines and optimizations can be employed. In Chapter 5, we showcase this advantage by proposing solutions targeting different categories of the applications. Overall, we classified the applications into four categories. In the first category, *cardinality of feature model* (i.e., $|\mathbb{CS}_{\mathbb{FM}}|$), applications depend on the number of all valid configurations induced by the system. For *cardinality of feature* (i.e., $|\mathbb{CS}_f|$), the number of valid configurations containing a certain feature is needed. The *cardinality of partial configuration* (i.e., $|\mathbb{CS}_{\mathbb{C}}|$) requires the number of valid configuration including some and excluding other features. *Cardinality of formula* (i.e., $|\mathbb{CS}_{\mathbb{FM}'}|$) refers to the number of valid configurations induced by the feature model under restrictions not covered by the other categories, such as only caring for specific features.

Definition 16 (Cardinality of a Feature Model)**.** The *cardinality of the feature model* $|\mathbb{CS}_{\mathbb{FM}}|$ denotes the number of valid configurations induced by the entire feature model (i.e., the cardinality of the configuration space $\mathbb{CS}_{\mathbb{FM}}$).

Definition 17 (Cardinality of a Feature)**.** The *cardinality* $|\mathbb{CS}_f|$ *of a feature* f denotes the number of valid configurations that include that feature (i.e., $|\mathbb{CS}_f| = |\{(I, E) \mid f \in I, (I, E) \in \mathbb{CS}\}|$).

Definition 18 (Cardinality of a Partial Configuration)**.** The *cardinality* $|\mathbb{CS}_{\mathbb{C}}|$ *of a partial configuration* $\mathbb{C} = (\mathbb{I}, \mathbb{E})$ denotes the number of valid configurations that include each feature $i \in \mathbb{I}$ and exclude each feature $e \in \mathbb{E}$ (i.e., $|\mathbb{CS}_{\mathbb{C}}| = |\{(I, E) \mid \mathbb{I} \subseteq I, \mathbb{E} \subseteq E, (I, E) \in \mathbb{CS}\}|$).

Definition 19 (Cardinality of a Formula)**.** The *cardinality of a formula* $|\mathbb{CS}_{\mathbb{FM}'}|$ denotes the number of valid configurations under an arbitrary adaptation $\mathbb{FM}' = (\mathbb{FS}', \mathbb{CT}')$ of the feature model (i.e., the cardinality of the adapted configuration space $|\mathbb{CS}_{\mathbb{FM}'}|$).

The last category, cardinality of formula, was not part of the original survey [A1], but added in our **AMAI'24** publication [A2C1]. Initially [A1], we classified the associated applications in the category cardinality of feature model, as we considered the adapted formula as new feature model. Later, we added the cardinality of formula to capture (1) the potentially added overhead and (2) further clarify the difference between the applications.

For detailed descriptions, examples, and computation instructions on all applications listed in Table *18*A, we refer to the **VaMoS'21** publication [A1]. In the following, we present one example for each of the categories when applied to our running example Figure *18*B.

With *variability reduction* [A1], we can examine the impact of changes to the configuration knowledge by comparing the number of valid configurations to numbers of previous versions. One use case is deliberately reducing the number of valid configurations to make the product line less costly to maintain. According to Bagheri et al. [12] and Kübler et al. [129] the number of valid configurations can be an effective indicator for the maintainability of the underlying product line. However, introducing changes that effectively reduce the number of valid configurations is not reliable without knowing the number of valid configurations before and after change. We assume in our example that the modeling expert considers two options for reducing

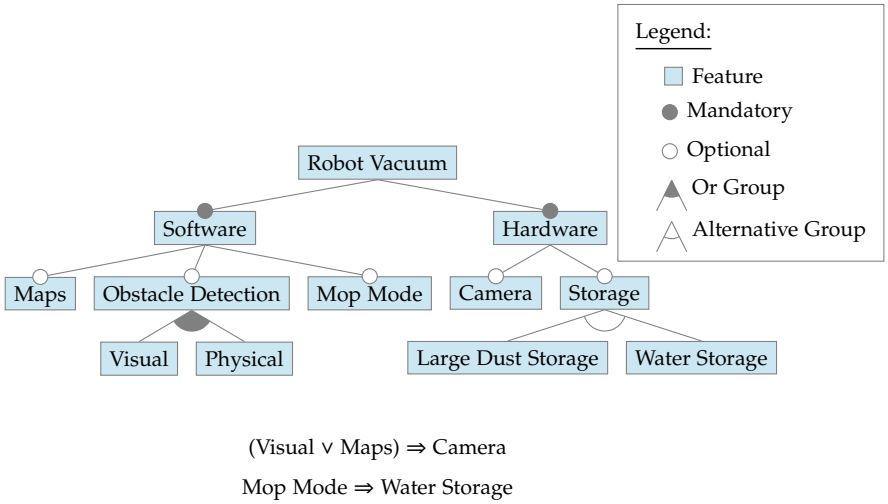


Figure 18b: *Running Example: Feature Diagram of Robot Vacuum*

the variability: setting `obstacle detection` to mandatory or adapting the first cross-tree constraint to `Visual ∨ Maps` $\iff$ `Camera`. Even for our small example, it is hard to manually estimate the effect on the number of valid configurations of both changes. Computing the number of valid configurations shows that enforcing `obstacle detection` provides a more effective reduction (i.e., 42 remaining valid configurations) compared to adapting the constraint as described (i.e., 48 remaining). The reduction in variability can also be used as sanity checks as an unexpected change in the number of valid configurations may indicate a modeling error.

With *feature prioritization* [A1], we can employ the number of valid configurations including a certain feature in the decision process. For instance, prioritizing a feature appearing in more valid configurations during development allows building more distinct products earlier. Considering our robot vacuum, the manager may have to decide whether to develop a `water storage` or a `large dust storage` first. Respectively, adding the features induces 40 and 30 valid configurations. Hence, developing `water storage` could be beneficial since more valid configurations depend on the availability of that feature.

Interactive configuration support [44, 146, A1] can help with configuring products which may require many decisions from the user. During configuration, the user often gets little guidance to understand the impact of decisions in popular online configurators [146]. Consequentially, the configuration process can be tedious and error-prone [44], especially considering that real-world product lines often contain thousands of features [118, 129, 145, B3]. Both, Mazo et al. [146] and Chen and Erwig [44] propose *optimizing the product derivation* by sorting feature decisions by the number of valid configurations considering the current partial configuration. The number of remaining valid configurations after a potential decision can also be directly shown to users to inform them about the impact of their decision. Depending on the user's preferences, they might favor selections that induce more or less flexibility, which is however complex or even infeasible to derive manually. For instance, suppose a user already added `mop mode` to their configuration, selecting `visual` leaves eight remaining valid configurations, while ten are left with `physical`. The number of

valid configurations under a certain partial assignment can also help with debugging as unexpected numbers may indicate issues in the model.

For many of the proposed applications, it may be helpful to consider the *variability of a feature subset* [96, A1]. Hereby, one is interested in the valid configurations over only a subset of features. For instance, in our running example, testing the hardware components may be considerably more expensive than testing software components. Hence, when examining the maintainability (e.g., the testing costs) the number of valid *hardware* configurations (8) may be more relevant compared to the overall number of valid configurations (60).

The variety of applications relying on configuration counting clearly indicates the potential benefits. Even the few examples indicate that configuration counting can provide insights for different stakeholders and product-line engineering tasks. Previously, the 12 available applications were spread over 9 publications (not counting duplicates). With our collection, the 21 applications including our new proposals can be accessed in one work [A1].

> 🔎 **Observation – RO1 (Applications & Problem Settings)**
>
> There are plenty of applications relying on configuration counting.

3.3 Problem Settings in Configuration Counting

In this section, we discuss our contributions that tackle the issue of *lacking computation instructions*. Our strategy of providing actionable computation instructions consists of two steps. First, we reduce the 21 applications to four types of counting operations [A2C1, A1]. Second, for each application, we present instructions on how to process the results of the counting operations for the specific applications [A1].

Transforming feature models to logic (mostly Boolean) and perform automated reasoning on the resulting formula is a de facto standard for feature-model analysis [16, 24, 52, 129, 136, 150, 174]. The main advantages of this approach are (1) many common representations of feature models can be straightforwardly translated to logic [21, 210] and (2) having logical representations enables using a plethora of tools that have been optimized over decades [78, 80]. Due to this synergy with logic, there is a natural reduction from configuration counting to model counting [129, 136]. Here, we can derive the number of valid configurations by computing the number of satisfying assignments on an equivalent formula.

Table 22A provides an overview on the reductions of the different categories of configuration counting to model counting. Here, $F_{\mathbb{FM}}$ is a formula semantically equivalent to the feature model $\mathbb{FM} = (\mathbb{FS}, \mathbb{CT})$, $f \in \mathbb{FS}$ is a feature, $\mathbb{C} = (\mathbb{I}, \mathbb{E})$ is a configuration that includes features $i \in \mathbb{I}$ and excludes features $e \in \mathbb{E}$, and $F_{\mathbb{FM}'}$ is an arbitrary adaptation of F_{FM}. The operation #SAT(F) computes the number of satisfying assignments given a formula F. As the original feature model $\mathbb{FM}$ and the corresponding formula $F_{\mathbb{FM}}$ are semantically equivalent, computing the model count of $F_{\mathbb{FM}}$ is equal to the cardinality of the feature model. To compute the cardinality of a feature, we can apply #SAT to $F_{\mathbb{FM}}$ and enforce the selection of f by conjuncting the literal f (i.e., $F_{\mathbb{FM}} \wedge f$). The same idea can be applied for

Table 22A: *Reducing Configuration Counting to Model Counting*

Category	Logical Operation
Cardinality of feature model $\lvert\mathbb{CS}_{\mathbb{FM}}\rvert$	$\#\mathrm{SAT}(F_{\mathbb{FM}})$
Cardinality of feature $\lvert\mathbb{CS}_{f}\rvert$	$\#\mathrm{SAT}(F_{\mathbb{FM}} \wedge f)$
Cardinality of partial configuration $\lvert\mathbb{CS}_{\mathbb{C}}\rvert$	$\#\mathrm{SAT}(F_{\mathbb{FM}} \wedge \bigwedge_{i \in \mathbb{I}} i \wedge \bigwedge_{e \in \mathbb{E}} \bar{e})$
Cardinality of formula $\lvert\mathbb{CS}_{\mathbb{FM}'}\rvert$	$\#\mathrm{SAT}(F_{\mathbb{FM}'})$

partial configurations. Here, for each included feature, a positive literal is conjuncted and for each excluded feature, a negative literal is conjuncted. The cardinality of formula heavily depends on the underlying application, but the general idea is to adapt the original formula depending on the use case and then use it as input for #SAT.

Note that it may make sense to further split the cardinality of formula into multiple categories if more applications are proposed. In its current state, the cardinality of formula covers a large variety of possible adaptations for which different reasoning engines may be beneficial. Even the two applications presented here, namely variability of feature subsets and rating errors, rely on different computational problems (cf. Section 5.1). Variability of feature subsets relies on projections [A2C1, C3] while rating errors depends on conjuncting additional constraints [129].

For each application we identified, the main computational effort lies in configuration-counting operations. Still, most (14 of 21) applications require further computational processing to acquire the results of interest. For instance, the *variability factor*, which describes the proportion of configurations that are valid, is computed with $\lvert\mathbb{CS}_{\mathbb{FM}}\rvert / 2^{\lvert\mathbb{FS}\rvert}$. We refer to the original work [A1] for instructions on further applications. Overall, our work provides actionable instructions for the single applications by combining the configuration counting operations shown in Table 22A with the application-specific computations.

> 🔎 **Observation – RO1 (Applications & Problem Settings)**
>
> Configuration-counting analyses can be reduced to model counting.

3.4 Computational Effort of Applications

In many cases, the computational effort of the applications is tied to the number of required counting queries [76, 161, A2C1] as even solving a single #SAT problem is computationally complex [31, 136, 218]. Consequentially, knowing the number of counting queries is vital to estimate the scalability of an application of interest and also to select promising reasoning engines. However, as discussed in Section 3.1, knowledge on the number of queries is often not explicitly available and requires interpretation of readers. In the **AMAI'24** publication [A2C1], we provide information on the number of required queries for analyses relying on different computational problems, such as SAT, #SAT, or AllSAT. In this thesis, we focus on analyses based on configuration counting (i.e., #SAT).

Table 23A shows the computational effort required for analyses based on configuration counting by considering two dimensions.[2] The *number*

Table 23A: *Number of Queries Required for Different Analyses*

Type	Analysis	Number of Queries	Repeatability
Card. of Feature Models	Variability Factor [26]	1	Static
	Variability Reduction [A1]	1	Static
	Degree of Orthogonality [51]	1	Static
	Maintainability Prediction [12]	1	Static
	Configuration Relevance [A1]	1	Static
	Cost Savings of Product Line [47]	1	Static
Card. of Features	Homogeneity [76]	$O(\|F\|)$	Static
	Atomic Set Candidates [A1]	$O(\|F\|)$	Static
	Feature Prioritization [A1]	$O(\|F\|)$	Static
	CTC Restrictiveness [A1]	$O(\|F\|)$	Static
	Degree of Reuse [49]	$O(\|F\|)$	Static
	Optimize Configuring [44, 146]	$O(\|F\|)$	Static
	Payoff Threshold [97]	$O(\|F\|)$	Static
Card. of Partial Conf.	Interactive Configuration [A1]	$O(\|F^2\|)$	Dynamic
	Uniform Random Sampling [161]	$O(\|F\|)$	Dynamic
	Rating Feature Interactions [A1]	$O(\|F^2\|)$	Dynamic
Card. of Formula	Rating Errors [129]	1	Dynamic
	Variability of Feature Subsets [A1]	1	Dynamic

2. Note that some analyses from Table *18A* are missing in Table *23A*. In our **AMAI'24** publication [A2C1], we moved few applications to a non-counting category because they are more naturally reduced to satisfiability problems.

of queries provides a worst-case estimation of the number of required operations to compute a result for the analysis once. *Repeatability* indicates whether performing the analysis may be useful more than once for the exact same feature model. For instance, the variability factor directly relies on the number of *all* valid configurations. Hence, repeating the analysis will always yield the same results. In contrast, providing counts during an interactive configuration process typically yields different results when performed multiple times. We refer to analyses with constant results as *static* and with potentially varying results as *dynamic*.

Analyses that depend on the *cardinality of the feature model* always require exactly one counting operation. For the *cardinality of features*, operations up-to the number of features are required. Some analyses like feature prioritization often require the cardinality of only some and, thus, only few counting operations. However, for instance, homogeneity by definition requires the count for every feature [76]. Still, for a fixed feature model, after computing the cardinality of all features once, no further counting queries are required, which makes each of the analyses static. During the interactive configuration process, it may be useful to compute the number of valid configurations for each remaining feature after every decision which results in a quadratic number of operations [A1]. Furthermore, computing the numbers can be useful for multiple configuration processes and, thus, we consider these analyses dynamic. Since there are exponentially many (i.e., $2^{\mathbb{F}\mathrm{S}}$) possible configurations, it is likely infeasible to (pre-)compute all combinations for real-world systems, as those often have thousands of features [129, 132, 145, B3]. Furthermore, we have observed industrial feature models with up-to 10^{1534} valid configurations [B1]. The analyses in the last two categories, namely *cardinality of partial configuration* and *cardinality of formula*, are typically valuable to execute multiple times for the same feature model and can produce exponentially many distinct counting operations.

Every counting operation is equivalent to solving a #SAT problem, which is #P-complete and generally considered to be harder than SAT [218]. Thus,

even applications relying on *a single* counting operation can be computationally expensive in practice. Most (14 out of 21) applications based on configuration counting typically require multiple counting operations. With real-world systems often having thousands of features [118, 145, B3], applications depending on partial configurations may induce millions of queries. For instance, considering a system with thousands of features, providing counts for every partial configuration along the way for *one* interactive configuration process can require millions of counting operations in the worst case. As solving #SAT is computationally expensive, the number of counting queries required for an analysis is crucial for judging the scalability.

Observation – RO1 (Applications & Problem Settings)

Many applications are dependent on numerous counting operations.

3.5 Summary

Our publications [A2C1, A1] on application scenarios on configuration counting lay the foundation for the remainder of this thesis. First, the collection of applications spanning over the whole development cycle indicates the *relevance* of configuration counting for product-line engineering and, thus, motivates the following contributions of this thesis. Second, the classification of counting types and number of queries helps to *streamline potential solutions* to focus on the identified problem settings.

All identified applications can be reduced to different model-counting operations [A2C1, A1]. In Chapter 4, we aim to examine the scalability of off-the-shelf tools for configuration counting. We empirically evaluate the performance of state-of-the-art model-counting solutions on industrial feature models. Another important observation is the reliance of most applications on a multitude of counting operations. In Chapter 5, we present different solutions tailored to the different applications we presented and in particular for scaling to myriads of counting operations.

4 *Evaluating the State of the Art on Configuration Counting*

1. Research Gap 26
2. Evaluate Performance 27
3. Summary 32

Research Objective 2 (Scalability State of the Art)

Central Research Question

How well does the state of the art scale for configuration counting?

Core Publications

EMSE'23 [B1]: "Evaluating State-of-the-Art #SAT Solvers on Industrial Configuration Spaces"
Chico Sundermann, Tobias Heß, Michael Nieke, Paul Maximilian Bittner, Jeffrey M. Young, Thomas Thüm, and Ina Schaefer, *Empirical Software Engineering (EMSE)*, 2023, Manuscript in Appendix C.3

VaMoS'20 [B2]: "Evaluating #SAT Solvers on Industrial Feature Models"
Chico Sundermann, Thomas Thüm, and Ina Schaefer, *Proc. Int'l Working Conf. on Variability Modelling of Software-Intensive Systems (VaMoS)*, 2020, Manuscript in Appendix C.4

SPLC'24 [B3]: "Collecting Feature Models from the Literature: A Comprehensive Dataset for Benchmarking"
Chico Sundermann, Vincenzo Francesco Brancaccio, Elias Kuiter, Sebastian Krieter, Tobias Heß, and Thomas Thüm, *Proc. Int'l Systems and Software Product Line Conf. (SPLC)*, 2024, Manuscript in Appendix C.5

Many applications rely on configuration counting [97, 129, A2C1, A1]. To use these applications in practice, scalable solutions for industrial feature models are required [28, 129, 163, 178]. Common feature analyses are typically realized by translating the feature model to propositional logic and then invoke off-the-shelf solvers [16, 24, 83, 123, 150, 161, 190]. Following this de facto standard, configuration counting can be reduced to propositional model counting [31, 129, 195]. However, model counting (#SAT for short) is computationally complex [31, 218]. Consequently, #SAT solvers may induce unacceptable runtimes when applied to available feature models. In this chapter, we discuss our contributions investigating the performance of off-the-shelf #SAT solvers for configuration counting [B1, B2]. With these contributions, we aim to (1) provide insights on the general scalability of #SAT for configuration counting, and (2) derive recommendations on promising solvers. As an indicator for general scalability, we evaluate the solvers for computing the cardinality of the entire feature model $|\mathbb{CS}_{\mathbb{FM}}|$. Orthogonal to our contributions to the evaluation of current technology for configuration counting, we performed a literature survey to collect industrial feature models to facilitate future empirical research [B3]. Motivated by efforts we put into collecting feature models for our evaluations on configuration counting [B1, B2], we provide a dataset that covers a wide range of application domains, structural properties, and complexities with respect to solver runtimes [B3].

This chapter is structured as follows. In Section 4.1, we discuss the research gap we identified regarding the performance of configuration counting on industrial instances. After providing insights into the relevance, we provide an overview of our evaluation of the state-of-the-art on configuration counting in Section 4.2 considering methodology, results, and potential limitations. Here, we discuss our contributions targeting the general scalability of configuration counting [B1], the scalability during evolution [B2], and our efforts to facilitate such evaluations in the future [A2C1]. Finally, we provide an overview on the major observations and their impact on this thesis in Section 4.3.

4.1 Research Gap

In this section, we discuss the research gap we identified regarding insights into the performance of off-the-shelf solutions. In Section 4.1.1, we discuss why examining potential solutions and in particular #SAT is relevant to apply configuration counting. In Section 4.1.2, we review the limitations of existing evaluations.

4.1.1 Relevancy

The applications presented in Chapter 3 can only be employed when scalable solutions are available. Each application depends on knowing the number of valid configurations, which is computationally complex to derive. Scalable solutions for such complex computations are a major challenge in practice [28, 96, 129, 132, 163, 178]. Even feature models limited to propositional constraints can represent any propositional formula [118]. Consequentially, configuration counting is at least as hard as solving #SAT problems, which is generally considered to be harder than SAT [31] (cf. Chapter 2 for more details on the complexity). Even though the worst-case time complexity is exponential, strategies in practice often scale to large formulas [40, 78, 133, 194, 215]. However, without identifying promising strategies and scalable tools, configuration counting may exceed an acceptable resource consumption (e.g., response time).

Model counting (#SAT) appears to be the most promising strategy for configuration counting [136, 174]. First, available industrial feature models are typically represented as logical formulas (most commonly CNF) or formats that can be translated to logic without exponential increase in size [29, 118, 131, 132, 156, 171, B3]. Hence, #SAT is a natural solution for configuration counting. Second, #SAT solvers have been optimized for decades [78, 182, 215] and can scale for instances with up-to millions of variables [78]. Third, while #SAT has not been directly compared to other techniques for model counting, evaluations considering other strategies for counting (e.g., enumerating the configurations [148, 174]) report considerably worse scalability [26, 98, 174] than reports considering #SAT [97, 129, 136, 161]. There were also some ad-hoc strategies that target counting on the feature tree (i.e., parent-child relationships from feature diagrams, cf. Section 2.1) [75, 76, 100]. However, the complexity of harder (w.r.t. to solver performance) industrial instances with a tree hierarchy typically lies within the cross-tree constraints [103, 118, B1], even worse many feature models considered in practice (e.g., Linux [132]) have no feature tree whatsoever [132, 156, 171, B3].

Selecting a suboptimal solver for a given use case may induce considerably increased runtimes, as there are often substantial performance differences between them [76, 78, 133, 152]. An empirical evaluation of #SAT solvers for configuration counting can be used to derive recommendations for promising solvers. However, without performance evaluation in the domain of interest, it is difficult to make an informed decision on promising solvers for that domain. First, performance from other domains does not necessarily transfer to feature models [7]. Second, there is a plethora of #SAT solvers available [10, 40, 78, 133, 182, 194, 215]. Here, an empirical evaluation on the performance of current #SAT solvers targeting industrial systems from different application domains can provide more accurate performance indicators for researchers and practitioners.

4.1.2 Limitations of Related Work

Evaluations examining the performance of solutions for configuration counting are limited regarding the *number of evaluated tools* [76, 129, 136]. Previous evaluations on feature models only considered at most two off-the-shelf #SAT solvers in their evaluation [76, 129, 136], while a collection of solvers from the model counting competition lists 18 solvers [78]. Furthermore, #SAT solvers are constantly updated and optimized [78, 119, 194] and the newest solvers have not been released for previous evaluations on feature models [76, 129, 136]. For instance, the first version of Ganak [194], which is one of the best performing solvers in recent model counting competitions,[1] was published in 2019, but the latest previous evaluation on the performance of configuration counting was in 2015 by Liang et al. [136].

1. Ganak was first in 2020 [78] and 2024 and second in 2021 and 2023 [77]

Previous evaluations only provide limited insights into feature models in practice [76, 96, 129, 136]. Fernández-Amorós et al. [76] use feature models from the SPLOT [149] database which are mostly academic, and the feature models contain only few features compared to larger industrial systems [29, 118, 121, 145, 156]. Kübler et al. [129] consider only confidential feature models from one application domain (i.e., automotive) in their evaluation. Liang et al. [136] consider only feature models from the systems software domain modelled in the Kconfig [50, 191] format. Overall, even when aggregating insights from the different evaluations [76, 96, 129, 136], only four[2] #SAT solvers and industrial systems from two[3] target domains have been considered.

2. The #SAT solvers previously evaluated are SharpSAT [136, 215] Cachet [76, 129, 182], c2d [54], and Relsat [18, 76].

3. Apart from SPLOT models [76, 149], the target domains were limited to operating systems [136] and automotive [96, 129].

4.2 Evaluating the Performance of Available Solutions

In the following, we give an overview on our contributions examining the performance of state-of-the-art solutions for configuration counting [B1, B2]. Note that we focus on the scalability of those solutions for computing the number of all valid configurations once as indicator for the general scalability of configuration counting. The problem settings we identified in Section 3.3 all base on highly similar formulas, which hints that the performance for the cardinality of the feature model is a reasonable first indicator. Furthermore, in our contributions presented in Chapter 5, we observe that feature models that are complex for computing the number of all valid configurations are generally complex across all considered applications [A2C1, C3, C2, C4]. We provide a short overview of the

methodology to gather relevant solvers and subject systems. Using the insights of the empirical evaluation, we discuss the general scalability of the evaluated solvers and derive recommendations on which solver to use. In Section 4.2.1, we examine the general performance of #SAT for configuration counting, while we examine the performance during the evolution of systems in Section 4.2.2. For more details on the empirical evaluation, we refer to the original work [B1, B2].

4.2.1 Model Counting on Industrial Systems

In this section, we discuss methodology, results, and limitations for our empirical evaluation on the performance of state-of-the-art #SAT solvers for configuration counting on industrial feature models [B1, B2].

Collecting Solvers With our methodology to gather tools, we aimed to provide a broad coverage of available #SAT solvers. Nevertheless, we employ the following restrictions regarding the solvers' usability: (1) the solver needs to be generally accessible, (2) the solver needs to support DIMACS as input (i.e., standard format for CNFs), and (3) the solver can be employed as standalone blackbox tool. To find the solvers, we (1) collected solvers and corresponding publications that have been considered in work on configuration counting [44, 76, 100, 129, 153, 159, 168, 174], and (2) performed forward and backward snowballing from the identified publications. During our search for solvers, a report on the model counting competition 2020 [78] was published that also included a list of available #SAT solvers. We used their list to verify and extend our collection. The lists shared eleven solvers, our list included four additional solvers, and their list included one additional solver. The similarities of both lists indicate that our list provides a reasonable coverage of #SAT solvers in practice. We added the missing solver from their list and three #SAT solvers that entered the model counting competition 2020 but were not published before. Overall, our collection of solvers contains 19 #SAT solvers, which is the largest collection ever empirically evaluated in the context of model counting. For a complete list and additional information on the solvers and underlying technologies, we refer to the original work [B1].

Collecting Subject Systems Following our goal of examining the performance of #SAT solvers for configuration counting in practice, we consider only *industrial* feature models. Furthermore, we aim to evaluate a wide range of application domains and structural properties (e.g., number of features and constraints). As subject systems, we gathered industrial feature models from several smaller collections that were used in previous empirical evaluations [118, 155, 169, 171]. In addition, we were provided access to feature models representing three automotive product lines from our industry partner. Overall, we gathered 130 feature models that represent variants of 15 industrial subject systems. These feature models come from four different domains and have 76–18,616 features and 14–10,321 cross-tree constraints.

Results Figure 29A shows the runtimes of all 19 #SAT solvers on the considered feature models. Note that for the CDL subsystems, we show the median runtime over all underlying feature models. Out of the 19

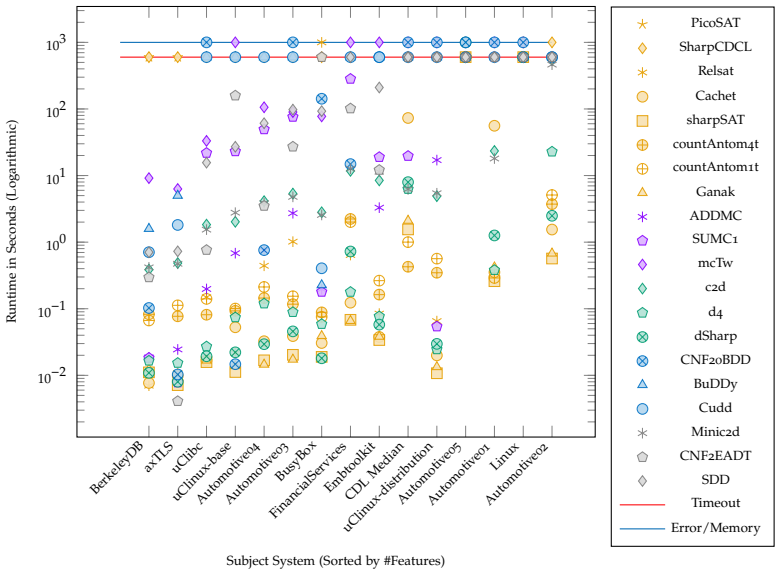


Figure 29A: *Runtime in Seconds for All Exact #SAT Solvers (Published in [B1] under CC-BY 4.0, https://creativecommons.org/licenses/by/4.0/)*

solvers, 8 solvers successfully evaluated 13 out of the 15 systems within the timeout of 10 minutes. The fastest five solvers, namely `sharpSAT`, `Ganak`, `countAntom`, `dSharp`, and `d4`, required less than a minute overall for these 13 systems. Furthermore, the fastest three solvers require less than 4 seconds for each of the 13 systems. In contrast, two solvers, namely `SharpCDCL` and `PicoSAT`, scaled to none of the systems.

🔎 Observation – RO2 (Scalability State of the Art)

Several #SAT solvers scale for many industrial feature models.

There are also two systems that could not be evaluated by any of the #SAT solvers within the timeout: a feature model describing the configuration space of the Linux kernel from 2010 (version 2.6.33) and the largest (w.r.t. number of features and constraints) confidential feature model from our industry partner. We repeated the experiment on these two systems for the eight solvers that successfully evaluated all remaining systems with a 24 hour timeout. Even with this substantially increased timeout, none of the solvers is able to compute a result for either of the two systems. In addition to the exact solvers, we evaluated also two approximate #SAT solvers [43, 225] which estimate the number of satisfying assignments. The approximate solvers also failed to compute a result within the timeout of ten minutes. In general, the approximate solvers scaled to fewer instances than the fastest exact solvers [B1].

🔎 Observation – RO2 (Scalability State of the Art)

For few industrial feature models, none of the solvers scale.

Limitations For a thorough discussion of all threats to validity we identified, we refer to the original publications [B1, B2]. Here, we focus on limitations regarding the transferability of our conclusions. The results

show that the considered solvers perform substantially differently on feature models. Hence, our results are not necessarily transferable to other and in particular new solvers. However, with our survey, we identified a plethora of #SAT solvers with our initial list even containing more solvers than the list of the model counting competition [78]. Furthermore, we also considered the solvers entering the competition that year. Within recent work, we also evaluated the top three performing #SAT solvers of the model counting competition 2023 which still failed for the same feature models [B3]. The performance on the considered feature models may also not be transferable to other systems or application domains. The main problem here lies in the availability of industrial feature models in the literature. To get a reasonable coverage, we gathered feature models used in various other empirical evaluations [29, 118]. The overall dataset included systems from various application domains and covers different structural properties.

To facilitate empirical research for future publications on feature models, we later performed a systematic literature review to collect publicly available industrial feature models [B3]. Here, we examined past publications that perform empirical evaluations on feature-model analysis and gathered industrial instances used in those. With our literature review, we overall collected 2,518 feature models representing different variants/versions of 41 systems. The feature models span over 13 application domains, have 101–80,258 features, and contain 0–767,040 constraints [B3]. In the original work [B3], we also observed that the collected feature models cover a wide range of evaluation times for SAT and #SAT. We provide a unified collection which includes the feature models in the original format, UVL [25] (i.e., a commonly employed format for feature modeling [70, 72, 144, 180, 205]), and DIMACS [175] (i.e., CNF standard format).[4] Furthermore, the collection contains various metrics describing the structure and semantics of the feature models and various options to derive subsets of the collection (e.g., models only from a specific application domain).

4. https://github.com/SoftVarE-Group/feature-model-benchmark

4.2.2 Model Counting during Evolution

To provide further insights into the scalability of configuration counting, we evaluated how the performance of #SAT solvers changes during the evolution of systems in our **VaMoS'20** publication [B2]. We had access to the histories of three automotive product lines by our industry partner. Here, each feature model represents one point in time during the evolution of the respective system. Figure 31A shows the runtimes of the #SAT solvers considered in the initial work for evaluating the feature models representing the systems' histories. For the first product line (i.e., Automotive03), there are too few data points to reasonably show an effect. The second product line (i.e., Automotive04) is rather easy to solve for every solver (i.e., runtimes of at most ten seconds), but the runtimes are increasing for every solver during the evolution. For Automotive05, the runtimes substantially increase over time. While the first ten evolution steps were evaluated within seconds by each of the three fastest solvers, overall only 33 (out of 136) evolution steps of Automotive05 could be evaluated within ten minutes. Within 24 hours, overall 62 evolution steps of Automotive05 could be successfully evaluated, which are all within the first 65 evolution steps. Note that the transferability here is rather limited as we only evaluated the histories from our industry partner. We discuss other similar observations from other work below in the part on *limitations*.

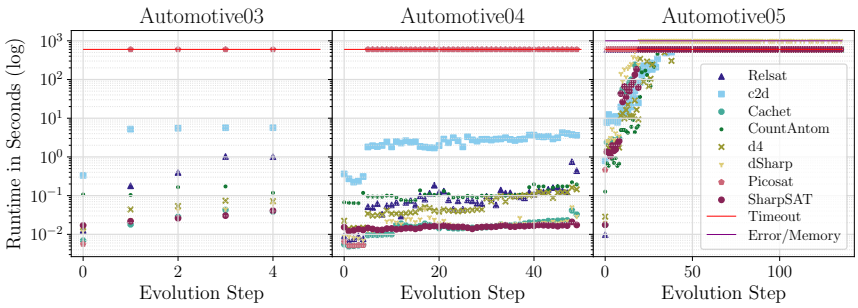


Figure 31A: *Runtime Automotive Histories*

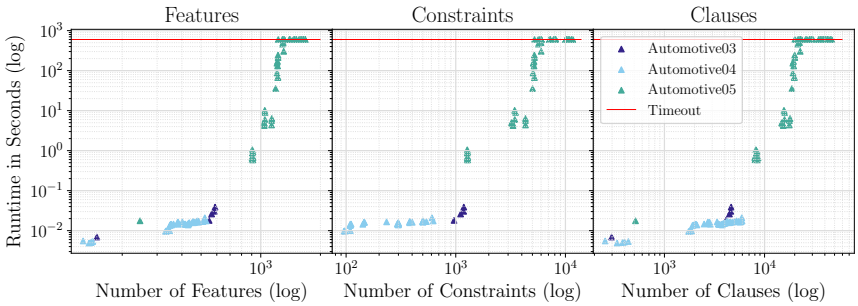


Figure 31B: *Structural Properties Automotive Histories*

Observation – RO2 (Scalability State of the Art)

Feature models often get harder to analyze over time.

For the considered histories, the feature models are growing over time. In particular, the number of features is weakly monotonically increasing for all three histories. The number of constraints, clauses, and valid configurations are also continuously increasing apart from very few exceptions. Figure 31B shows the correlation between structural metrics (i.e., number of features, constraints, and clauses) and the runtimes of the #SAT solvers. The visualization already shows a clear trend that larger feature models (w.r.t. to those metrics) are harder to analyze. In fact, there are strong positive correlations between the runtime and each of the three metrics.

Limitations For our examination of histories, the scope of considered subject systems is even narrower compared to the insights in Section 4.2.1. The three histories are limited to one domain (i.e., automotive) and even only one company. Hence, with the available data, we cannot know if the results transfer to other systems or application domains. However, we found indications that feature models commonly become more complex over time. In our work focusing on the Linux kernel, we also found that #SAT runtimes substantially increase over time for Linux [132]. Publications analyzing the history of systems also observed that the size of systems increases over time [145, 156, 171, B3]. Furthermore, examining the histories we gathered in our feature model collection [B3], we found that the number of features increase for all but one history (i.e., uClibc [171]).

4.3 Summary

In this chapter, we present our contributions [B3, B1, B2] that target empirically examining the performance of applying state-of-the-art solvers for configuration counting. We focus on #SAT solving as we found multiple indicators that #SAT is the most promising available strategy (cf. Section 4.1). For our empirical evaluation, we collected a plethora of #SAT solvers to provide a broad overview on the state of the art and evaluated them on industrial feature models from different domains. The results indicate that the best performing #SAT solvers scale to most industrial feature models.

Even though #SAT solving can generally enable configuration counting, we also found some challenges for their scalability. First, we identified two feature models that could not be analyzed with any #SAT solver. Feature models also often become even more complex during the evolution of a system. Second, many applications presented in Chapter 3 rely on numerous counting operations. Thousands or even millions of operations, each requiring a few seconds, are computationally demanding. Third, we were able to only identify scalable solutions for propositional logic, but many feature-modeling dialects include constructs that increase exponentially in size when represented in propositional logic. In Chapter 5, we discuss our contributions that tackle the described challenges.

5 *Accelerating Configuration Counting with Knowledge Compilation*

1. Research Gap 34
2. Knowledge Compilation for Feature-Model Analysis 35
3. Repetitive Queries with d-DNNFs 40
4. Projected d-DNNF Compilation 44
5. Pseudo-Boolean d-DNNF Compilation 49
6. Summary 55

Research Objective 3 (Accelerate the Multitude of Queries)

Central Research Question

How can we scale to the numerous different computations required for configuration counting?

Core Publications

AMAI'24 [A2C1]: "On the Benefits of Knowledge Compilation for Feature-Model Analyses"
Chico Sundermann, Elias Kuiter, Tobias Heß, Heiko Raab, Sebastian Krieter, and Thomas Thüm, *Annals of Mathematics and Artificial Intelligence (AMAI)*, 2024, Manuscript in Appendix C.2

TOSEM'24 [C2]: "Reusing d-DNNFs for Efficient Feature-Model Counting"
Chico Sundermann, Heiko Raab, Tobias Heß, Thomas Thüm, and Ina Schaefer, *Trans. on Software Engineering and Methodology (TOSEM)*, 2024, Manuscript in Appendix C.6

ASE'24 [C3]: "Efficient Slicing of Feature Models via Projected d-DNNF Compilation"
Chico Sundermann, Jacob Loth, and Thomas Thüm, *Proc. Int'l Conf. on Automated Software Engineering (ASE)*, 2024, Manuscript in Appendix C.7

TSE Sub. [C4]: *Pseudo-Boolean d-DNNF Compilation for Expressive Feature Modeling Constructs*
Chico Sundermann, Stefan Vill, Elias Kuiter, Sebastian Krieter, Thüm, and Matthias Tichy, Submitted to *IEEE Trans. on Software Engineering (TSE)*, 2025, Manuscript in Appendix C.8

Applications of configuration counting depend on solving many computationally complex operations [A2C1, A1]. These counting operations rely on different parameters, such as targeting only valid configurations containing a certain feature [47, 97, 100, A2C1]. Depending on the concrete application, using off-the-shelf solutions, such as black-box #SAT solvers [40, 194, 215], may not scale even if computing the number of valid configurations for a system can be performed once in acceptable response time. In this chapter, we tackle the scalability of different applications in configuration counting with tailored solutions for different problem settings. The majority of applications dependent on configuration counting have in common that they require *numerous counting queries* [A2C1] (cf. Section 3.4). Hence, our overall strategy to tackle the scalability of configuration counting revolves around *knowledge compilation*, which translates the feature model to a beneficial format (w.r.t. answering multiple queries efficiently). Here, we find that d-DNNFs are in a sweet spot for configuration counting regarding compilation time and enabled counting operations [A2C1]. We develop optimized algorithms for querying d-DNNFs, revealing the substantial runtime benefits of *having* a d-DNNF for configuration counting [C2]. Consequently, instead of employing single-shot computations for the other problem settings, we also target

d-DNNFs with two novel *compilation* strategies. We facilitate configuration counting on feature subsets by employing projected d-DNNF compilation [C3]. Furthermore, we extend d-DNNF compilation to handle feature models containing constructs that are not well suited for propositional encoding [C4]. Overall, with our contributions, we tackle each category of applications presented in Section 3.3.

This chapter is structured as follows. In Section 5.1, we discuss the need for accelerating configuration counting based on the observations from Chapter 3 and Chapter 4. Furthermore, we provide a mapping between problem settings and our proposals to tackle them. Towards our goal of applying knowledge compilation for configuration counting, we examine promising target languages in Section 5.2. With the identified sweet spot d-DNNFs as target language, we provide concrete realizations for the counting operations and further showcase the advantages in Section 5.3. Then, we extend the applicability of d-DNNFs with two novel compilation approaches targeting (1) variability of feature subsets in Section 5.4, and (2) expressive feature-model dialects in Section 5.5. Finally, we provide a wrap up on the advances for configuration counting by the described contributions in Section 5.6.

5.1 Research Gap

In this chapter, we tackle the research gap of missing scalable solutions for various problem settings in configuration counting. We discuss the relevance and limitations of related work separately for the problem settings tackled by the different contributions in Section 5.2, Section 5.3, Section 5.4, and Section 5.5. In this section, we provide a mapping between the core contributions considered in this chapter and the applications accelerated by them. As observed in Chapter 4, available solvers scale to most feature models [B1, B2], but may not be suitable for the variety of counting queries required for different applications. Our contributions in this chapter target the scalability of configuration counting. To this end, we propose multiple strategies tailored to the different requirements in configuration counting. Our contributions focus on specific categories of applications [A2C1, C3, C2] or specific feature-modeling constructs [C4].

Table 35A provides an overview on the applications covered by the respective publications [A2C1, C3, C2, C4]. Here, we distinguish whether the strategy specifically targets an application (indicated by ●), does not focus on this task but still yields benefits (indicated by ○), or provides no direct benefits. With our first contribution (cf. Section 5.2), we examine the suitability of knowledge-compilation target languages for different applications [A2C1]. Considering the tractable analyses over the considered target languages, the insights from our survey are relevant for all applications dependent on multiple counting operations or adaptations of the original formula (i.e., $|\mathbb{CS}_f|$, $|\mathbb{CS}_C|$, and $|\mathbb{CS}_{\mathbb{FM}'}|$). In the second contribution (cf. Section 5.3), we employ d-DNNFs for repetitive queries targeting the cardinality of features (i.e., $|\mathbb{CS}_f|$) and the cardinality of partial configurations (i.e., $|\mathbb{CS}_C|$) [C2]. When a corresponding d-DNNF is already available, the d-DNNF can also be used for the cardinality of feature model (i.e., $|\mathbb{CS}_{\mathbb{FM}}|$). Analogously, if a d-DNNF based on an adapted formula (i.e., $|\mathbb{CS}_{\mathbb{FM}'}|$) is already available, it may also make sense to employ applications facilitated by reusing d-DNNFs. With our third contribution [C3]

Table 35A: *Mapping Between Core Contributions and Tackled Applications*

Application	Sections			
	5.2 [A2C1] AMAI'24	5.3 [C2] TOSEM'24	5.4 [C3] ASE'24	5.5 [C4] TSE Sub.
Cardinality of feature model $\lvert\mathbb{CS}_{\mathbb{FM}}\rvert$	○	○	○	●
Variability factor	○	○	○	●
Degree of orthogonality	○	○	○	●
Cost savings of a product line	○	○	○	●
Void feature model	○	○	○	●
Maintainability prediction	○	○	○	●
Variability reduction	○	○	○	●
Configuration relevance	○	○	○	●
Cardinality of feature $\lvert\mathbb{CS}_{f}\rvert$	●	●	○	○
Degree of reuse	●	●	○	○
Homogeneity	●	●	○	○
Core, dead & false-optional	●	●	○	○
Payoff threshold of a feature	●	●	○	○
Optimize configuration process	●	●	○	○
Atomic set candidates	●	●	○	○
Feature prioritization	●	●	○	○
CTC restrictiveness for features	●	●	○	○
Cardinality of partial configuration $\lvert\mathbb{CS}_{\mathbb{C}}\rvert$	●	●	○	○
Uniform random sampling	●	●	○	○
Atomic sets	●	●	○	○
Rating feature interactions	●	●	○	○
Interactive configuration support	●	●	○	○
Cardinality of formula $\lvert\mathbb{CS}_{\mathbb{FM}'}\rvert$				
Variability of feature subsets	●	○	●	○
Rating errors	●	○		○

(cf. Section 5.4), we propose projected d-DNNF compilation which targets the *variability of feature subsets* (cf. $\lvert\mathbb{CS}_{\mathbb{FM}'}\rvert$in Table 35A). As the compiled d-DNNF can be further reused with our **TOSEM'24** publication [C2], the contribution is also relevant for those cardinality categories. Last, in our fourth contribution [C4] of this chapter (cf. Section 5.5), we propose pseudo-Boolean d-DNNF compilation which enables representation and d-DNNF compilation of more advanced feature-modeling constructs [C4] and, thus, can be useful for all types of cardinalities (i.e., $\lvert\mathbb{CS}_{\mathbb{FM}}\rvert$, $\lvert\mathbb{CS}_{f}\rvert$, $\lvert\mathbb{CS}_{\mathbb{C}}\rvert$, and $\lvert\mathbb{CS}_{\mathbb{FM}'}\rvert$). Overall, with our four contributions, we tackle the scalability of every application presented in Chapter 3.

5.2 Knowledge Compilation for Feature-Model Analysis

In this section, we present further contributions of our **AMAI'24** publication [A2C1]. We already covered the first part of the original publication that examines the different feature-model analyses and underlying computational problems in Chapter 3. Now, we discuss our contributions that examine knowledge compilation target languages with respect to their applicability for different analyses and the compilation effort.

5.2.1 Targeted Problem

Knowledge compilation (cf. Section 2.2.3) seems promising for feature-model analysis due to the plethora of analyses dependent on solving complex problems on logical formulas [24, 76, 100, 128, 168, 193, A2C1, A1]. While *ordered binary decision diagrams (OBDDs)* have been considered for feature-model analysis already a decade ago [4, 64, 103, 168], the exploration of other knowledge compilation target languages is limited. The only other considered language appears to be d-DNNF which has attracted attention over the last years [36, 193, C2]. In our **AMAI'24** publication [A2C1], in addition to the contributions described in Section 3.4, we explore (1) which knowledge compilation languages are available, (2) their capabilities w.r.t. tractable feature-model analyses, (3) their *succinctness*, and (4) the performance of available knowledge compilers.

Relevance Many feature-model analyses may benefit from knowledge compilation [24, 76, 97, 127, 193, A2C1]. Every application relying on the cardinality of feature, cardinality of partial configuration, or cardinality of formula may require solving multiple counting problems on the same feature model [A2C1, A1]. Furthermore, adaptations of the feature model required for the cardinality of formula $|\mathbb{CS}_{\mathbb{FM}'}|$ can also be facilitated by different properties of knowledge compilation languages, such as poly-time conjunction [56]. From the surveyed applications shown in Table 35A, 14 out of 21 applications belong to these categories (i.e., $|\mathbb{CS}_f|$, $|\mathbb{CS}_C|$, and $|\mathbb{CS}_{\mathbb{FM}'}|$) and, thus, may benefit from knowledge compilation.

There are various dimensions to consider when searching for the most promising knowledge compilation target language [56]. Typically, for a given use case, one is interested in the target language that requires the least computational effort to compile but still supports poly-time queries for all operations of interest [56]. Selecting an unsuitable target language may have substantial implications on the performance if relevant queries are not poly-time on that language or the compilation is computationally expensive. However, to make an informed decision on promising knowledge compilation target languages, information on several complex aspects is required. First, the computational operations required for the analysis of interest must be known. Second, an overview of available target languages and capabilities w.r.t. tractable operations is needed. Third, the efficiency of target languages regarding compilation and reasoning is essential to evaluate their potential performance benefits. Currently, deriving knowledge on each of those aspects requires *effort* to collect the relevant information and *expertise* to combine the gathered insights.

Application in Practice As indicated by several surveys on industrial practices with product-line engineering [28, 163], scalable solutions for analyzing configuration knowledge are mandated. In other publications from an industrial context [94, 129], the authors describe concrete applications that could be facilitated by knowledge compilation. Kübler et al. [129] discuss configuration counting employed in the context of Mercedes-Benz (i.e., automotive manufacturing) where they propose applications dependent on all four types of cardinality (i.e., $|\mathbb{CS}_{\mathbb{FM}}|$, $|\mathbb{CS}_f|$, $|\mathbb{CS}_C|$, and $|\mathbb{CS}_{\mathbb{FM}'}|$). Within industrial collaborations, we employed t-wise sampling [94] and the variability of feature subsets [96] (cf. Table 35A), which can both benefit from knowledge compilation [56, A2C1]. Furthermore, we were also often

confronted with scalability issues in industry collaborations that could be remedied with knowledge compilation. From our experience, a major issue in practice is identifying scalable reasoning engines for a given problem setting without extensive expertise.

Limitations of Related Work We are not aware of any work that compares knowledge compilation languages for feature-model analysis. Several publications consider OBDDs [4, 64, 103, 168] and some publications consider d-DNNFs [36, 193], but neither of these publications consider multiple languages. In our contribution on evaluating #SAT solvers [B1, B2], the #SAT solvers include knowledge compilers for multiple languages, namely OBDD, d-DNNF, and sentential decision diagrams [165], but we evaluate them only as black-box model counters.

With their knowledge compilation map, Darwiche and Marquis [56] also consider a variety of knowledge compilation languages. Part of our work [A2C1] is heavily inspired by their approach on examining knowledge compilation languages. In particular, they present tractable operations and succinctness for the languages and lay the foundations for our further examinations. However, as the work from Darwiche and Marquis [56] is from 2002, several modern languages are missing. In addition, without expertise it may be hard to derive a suitable knowledge compilation language for a *feature-model* analysis of interest. Darwiche and Marquis [56] also limit their examination to theoretical analysis of the languages and do not consider an empirical evaluation of existing knowledge compilers. Fargier and Marquis [71] provide an extension for the knowledge compilation map in 2008, but there are still some languages missing (e.g., sentential decision diagrams [165]) and their examinations also remain theoretical. In general, to the best of our knowledge, our **AMAI'24** publication [A2C1] is the first work to compare knowledge compilation to different target languages *empirically*. For further related work in the scope of knowledge compilation for feature models and surveys on feature-model analyses, we refer to the **AMAI'24** publication [A2C1].

5.2.2 Properties of Target Languages

In this section, we provide insights on our theoretical analysis of knowledge compilation target languages for feature-model analysis. We (1) present our methodology to gather target languages and (2) provide a mapping between feature-model analyses and target languages they are *tractable* for.

Survey Methodology We give a high-level overview of our methodology for gathering knowledge compilation target languages and knowledge compilers in Figure 38A. To gather target languages, we performed a systematic literature survey. Here, we also search for tools (i.e., knowledge compilers) used or mentioned in identified publications. Afterward, we conducted a GitHub search for knowledge compilation and all languages identified in the literature survey. Overall, we identified ten[1] knowledge compilation target languages and ten publicly available knowledge compilers. For details on the methodology (e.g., applied search criteria), we refer to the **AMAI'24** publication [A2C1].

1. Note that several languages have subclasses which we discuss in the original work [A2C1]

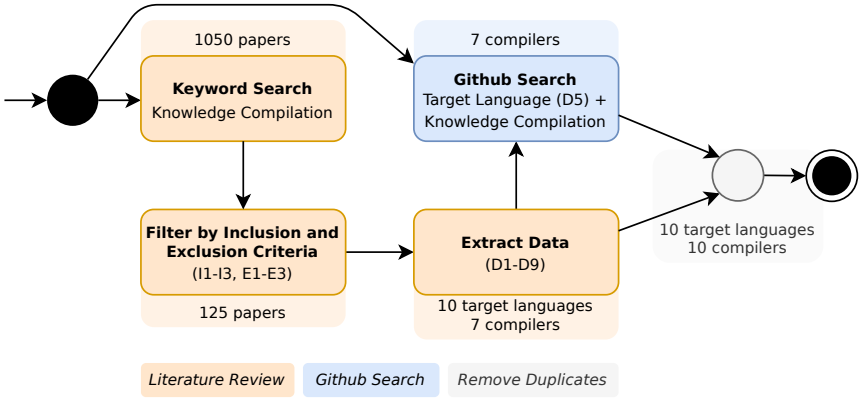


Figure 38A: *Methodology for Detecting Target Languages and Knowledge Compilers (Published in [A2C1] under CC-BY 4.0, https://creativecommons.org/licenses/by/4.0/)*

Table 38A: *Tractability of Feature-Model Analyses for Knowledge-Compilation Target Languages (Adapted from [A2C1], licensed under CC-BY 4.0, https://creativecommons.org/licenses/by/4.0/)*

	SAT-Based			#SAT-Based			AllSAT	Algebraic		
	CO	CO_f	CO_C	$\lvert\mathbb{CS}_{\mathbb{FM}}\rvert$	$\lvert\mathbb{CS}_f\rvert$	$\lvert\mathbb{CS}_C\rvert$	Enum.	Comp.	Diff.	Proj.
CNF										
Horn	✓	✓	✓							
EPCCL	✓	✓	✓							
DNF	✓	✓	✓				✓			✓
DNNF	✓	✓	✓				✓			✓
d-DNNF	✓	✓	✓	✓	✓	✓	✓			
EADT	✓	✓	✓	✓	✓	✓	✓			
SDD	✓	✓	✓	✓	✓	✓	✓			
PI	✓	✓	✓	✓	✓	✓	✓			✓
IP	✓	✓	✓	✓	✓	✓	✓			
ROBDD	✓	✓	✓	✓	✓	✓	✓	✓	✓	
MODS	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓

Mapping Feature-Model Analyses and Suitable Target Languages To guide researchers and practitioners when selecting promising knowledge compilation target languages, we provide a mapping between knowledge compilation target languages and feature-model analyses that are tractable for the languages. The mapping is shown in Table 38A. Here, a tractable analysis is denoted with a checkmark ✓. For instance, reduced ordered binary decision diagrams (ROBDD) supports poly-time queries (w.r.t. its number of nodes) for all feature-model analyses, except analyses based on slicing (also called projection [9] or forgetting [56]) [56, A2C1]. To derive the mapping, we combined our insights on logical operations required for feature-model analyses and the tractability of those logical operations on the target languages [A2C1]. We collected information on the latter from the knowledge compilation map [56] and publications corresponding to languages missing in the original map [63, 120, 219, 223]. CNF is included for comparison as it is used as source language baseline (cf. Section 2.2.3) by Darwiche and Marquis [56] and in our mapping.

In our **AMAI'24** publication [A2C1], we also consider *succinctness* [56] as further measure to derive the most promising target language. A

language A is more succinct than another language B if there is a poly-time and poly-space conversion from A to B, but not the other way around. Darwiche and Marquis [56] propose to select the most succinct target language for which the analysis of interest is tractable. We also examine the succinctness of all languages in our work [A2C1]. However, while succinctness can be a valuable initial indicator, scalability in practice depends on the performance of available knowledge compilers. Hence, in this thesis, we omit our insights on succinctness and focus on the empirical results.

5.2.3 Empirical Evaluation

In our empirical evaluation, we examined the performance regarding compilation time on industrial feature models for every collected knowledge compiler.

Setup With our survey, we identified ten knowledge compilers: three for d-DNNFs [54, 133, 152], three for sentential decision diagrams (SDDs) [55, 157, 165], three for OBDDs [141, 197, 216], and one for extended affine decision trees (EADTs) [120]. As subject systems, we consider 53 industrial feature models used in previous empirical evaluations on feature-model analysis [118, 155, 161, 170, 171]. These feature models come from 13 different domains, have 11–62,482 variables, and 1–350,221 constraints. We invoke each compiler on every feature model and measure the compilation times.

Results Figure 40A shows the runtimes of all considered knowledge compilers on the 53 industrial feature models. Six feature models could not be compiled to any target language. Two of the three d-DNNF compilers scaled to the remaining 47 industrial feature models. One SDD compiler (`MiniC2D`) scaled to 46 feature models. Other compilers scaled for at most 29 feature models. In particular, for only 28 feature models we could compile an OBDD (three compilers overall) and for 25 an extended affine decision tree (one compiler) [120].

Overall, our insights imply that d-DNNFs hit a sweet spot regarding the tractability for configuration counting and compilation performance [A2C1]. In particular, d-DNNFs enable linear runtime queries w.r.t. their size for the cardinality of feature models, cardinality of features, and cardinality of partial configurations. For the target languages, for which we found a knowledge compiler, d-DNNFs compiler scaled to the highest number of feature models. Nevertheless, there are limitations of the applicability of d-DNNFs as algebraic operations are not supported which are relevant for different feature-model analyses, such as some incarnations of cardinality of formula $|\mathbb{CS}_{\mathbb{FM}'}|$. Here, OBDDs or reduced OBDDs could be valuable even though the available compilers scaled considerably worse than the d-DNNF compilers.

Observation – RO3 (Accelerate the Multitude of Queries)

For configuration counting, d-DNNF appears to be a sweet spot between tractability of relevant operations and compilation performance.

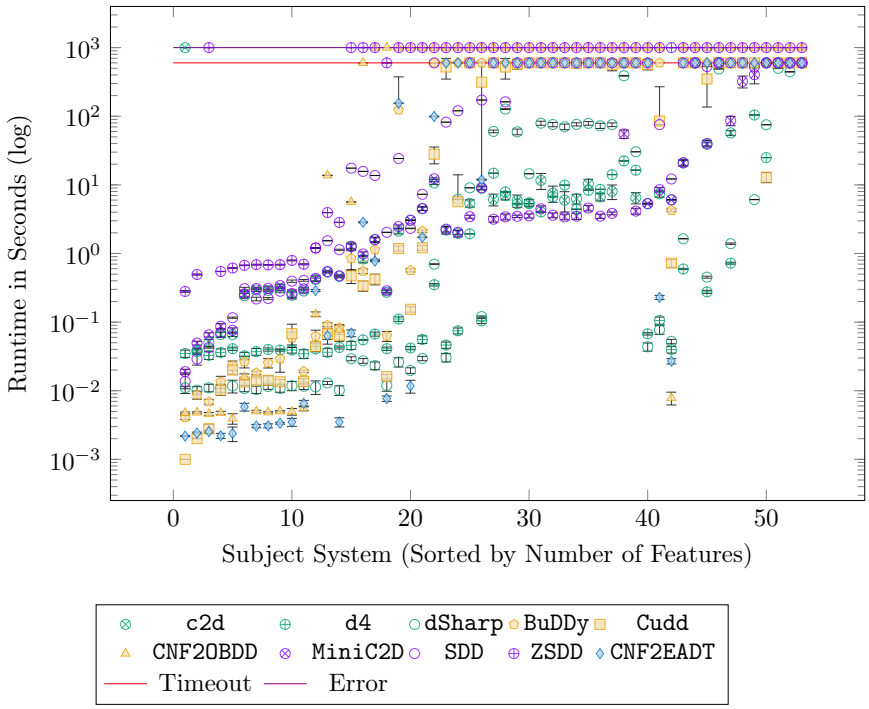


Figure 40A: *Runtimes Knowledge Compilers (Published in [A2C1] under CC-BY 4.0, https://creativecommons.org/licenses/by/4.0/)*

Limitations For a detailed discussion on potential threats to validity, we refer to the **AMAI'24** publication [A2C1]. One important limitation to consider is the gap between considered knowledge compilation target languages and tool support. While we identified ten target languages with even more subclasses, we only found knowledge compilers for four of those languages. Furthermore, the considerable difference between performances of compilers for the same languages indicate that the concrete realization is very relevant for the compilation performance. Hence, once more compilers for other formats are available, other target languages may become more promising. For instance, after our **AMAI'24** publication [A2C1] has been published, there has been advances in OBDD compilation for feature models which may motivate employing OBDDs in configuration counting [64, 102].

5.3 Repetitive Queries with d-DNNFs

In this section, we discuss the contributions presented in our **TOSEM'24** publication [C2] on reusing d-DNNFs for repetitive counting queries.

5.3.1 Targeted Problem

With our **TOSEM'24** publication [C2], we tackle feature-model analyses relying on cardinalities of features $|\mathbb{CS}_f|$ and cardinalities of partial configurations $|\mathbb{CS}_C|$.

Relevance Many feature-model analyses are dependent on numerous counting queries [76, A2C1, A1]. Out of 21 applications in our survey,

12 applications rely on the cardinality of features or cardinality of partial configurations (cf. Section 3.3). Our observations in Chapter 4 revealed that the state-of-the-art approach (i.e., black-box #SAT solving) scales for most feature models, but the fastest solver still often requires few seconds for those. Having thousands or even millions of counting queries may amount to unacceptable runtimes with black-box #SAT solving. Hence, to enable these analyses in practice, more efficient reasoning is required.

Application in Practice As discussed already in Section 5.2, scalable reasoning is demanded in practice [28, 163]. While concrete applications are typically not considered in industrial case studies [27, 28, 163, 224], Kübler et al. [129] employ applications dependent the cardinality of features $|\mathbb{CS}_f|$ and the cardinality of partial configurations $|\mathbb{CS}_\mathbb{C}|$ for Mercedes-Benz product lines. The problem setting targeted in this section (i.e., repetitive counting queries on the same feature model) is motivated also by our industrial collaborations. In particular, we faced major scalability issues with the prior approach of using black-box #SAT solvers for computing the cardinality of features. Within this industry project, we integrated our proposal of *reusing d-DNNFs* for those computations and substantially reduced the scalability issues.

Limitations of Related Work There is a lack of efficient solutions for repetitive queries of configuration counting. The vast majority of publications presenting approaches for counting only consider one-shot invocations of #SAT solvers for the entire feature model (i.e., $|\mathbb{CS}_{\mathbb{FM}}|$) [96, 129, 132, 136]. While repetitive invocations of #SAT can be employed for tasks depending on multiple queries [97, A2C1, A1], the solver has to start from scratch for every query (e.g., for every feature cardinality). Fernández-Amorós et al. [76] propose an ad-hoc solution for computing the commonality (i.e., relative cardinality) of features. However, their proposal (1) is limited to the cardinality of features and (2) focuses heavily on hierarchical constraints. As more thoroughly discussed in Section 4.2, scalability in practice mostly depends on cross-tree (i.e., non-hierarchical) constraints [100, 103, B1, B2].

In work focusing on d-DNNFs, concrete realization and empirical evaluations have been limited to compilation [54, 133, 152]. Darwiche [53] proposes two algorithms for counting on d-DNNFs which we base our approach on. However, he (1) proposes no further optimizations which are vital (as indicated by an ablation study we performed [209]), (2) performs no empirical evaluation, and (3) provides no implementation of their algorithms.

5.3.2 *Efficient d-DNNF Queries for Configuration Counting*

Given a d-DNNF, the model count can be computed by traversing the d-DNNF and applying rules based on the operator type [53]. Figure 42A shows counting queries on an example d-DNNF. The black numbers represent the number of all satisfying assignments. First, a *literal* induces one satisfying assignment. Second, for a *deterministic disjunction*, the number of satisfying assignments is the sum of the number for its disjuncts. As the disjuncts share no satisfying assignments, one could merge their solutions to derive the solutions for the disjunction without introducing any duplicates. Third, for a *decomposable conjunction*, the number of satisfying assignments is the product of the number for its conjuncts. As the

conjuncts share no variables, one could build the cross-product of their solutions to derive the solutions for the conjunction. The basic idea of our approach for the cardinality of features and partial configurations depends on *conditioning* the formula (i.e., assigning variables) and then traversing the d-DNNF again. In particular, we adjust the value of every literal to zero if it conflicts with the conditioning. For instance, for the partial configuration $\{\bar{c}, d\}$ in Figure 42A we set the literals c and $\bar{d}$ to zero.

Optimizations With the basic algorithm, the whole d-DNNF needs to be traversed for every query. As the d-DNNF represents the trace of an exhaustive DPLL traversal, we essentially perform a regular #SAT invocation minus some overhead for decision heuristics for every query. To improve the scalability, we adopted an alternative algorithm by Darwiche [53] for cardinality of features, namely partial derivatives.[2] Furthermore, we propose several optimizations listed in the following that reduce (1) the number of required traversals and (2) the share of a d-DNNF that needs to be traversed. First, we propose a constant time algorithm to detect core and dead features. Then, we derive model counts for queries implied by identified core and dead features. For instance, queries that set a dead feature to ⊤ always result in a model count of zero. Second, for every query, we identify which parts of the d-DNNF have changed and only traverse changed parts. Third, we provide multiple technical optimizations considering the data structures, arithmetic operations, and the traversal. For details on the optimizations, we refer to the original work [C2].

2. Partial derivatives are not applicable for conditioning more than one variable and, thus, not suitable for cardinality of partial configurations.

Tool Support Our approach for reusing d-DNNFs is publicly available in our d-DNNF reasoner `ddnnife`.[3] The reasoner supports various logical operations relevant for feature-model analysis, namely conditioning. checking satisfiability, counting, t-wise sampling [123, 138], uniform random sampling [193], deriving optimal configurations [179], atomic sets [190], and incremental adaptation of d-DNNFs [177]. `ddnnife` can be either used for one-shot computations or interactively with subsequent queries. We also integrated `d4` to support CNF as input which is then compiled to d-DNNF beforehand. While the tool is written in Rust, we provide native interfaces for Kotlin, which also enables usage in Java, and Python.

3. `https://github.com/SoftVarE-Group/d-dnnf-reasoner`

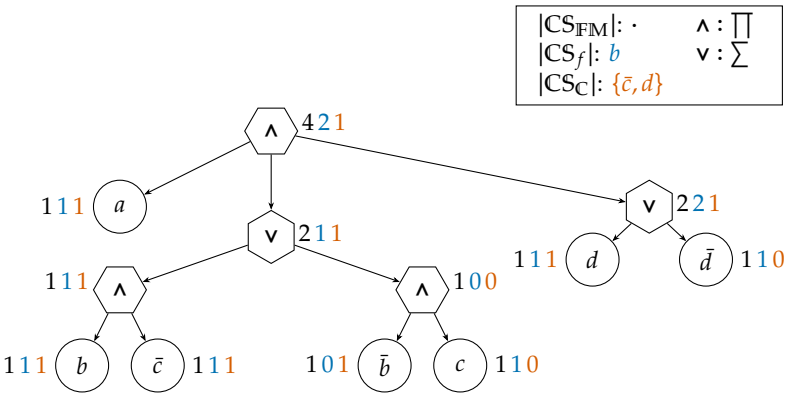


Figure 42A: *Computing Cardinalities on a d-DNNF (Adapted from [C2], licensed under CC-BY 4.0, `https://creativecommons.org/licenses/by/4.0/`)*

5.3.3 Evaluation

Setup As baseline for the state of the art, we use the three best performing #SAT solvers for feature models [B1] (cf. Chapter 4), namely `sharpSAT` [215], `countAntom` [40], and `Ganak` [194]. For d-DNNF compilation, we consider all d-DNNF compilers we are aware of [78, A2C1], namely `c2d` [54], `dSharp` [152], and `d4` [133]. While there is no publication mentioning a d-DNNF reasoner, we found a tool `query-ddnnf` on the homepage of researchers involved in the development of `d4`.[4] However, it appears that `query-ddnnf` has not been evaluated or even considered in another scientific publication. We evaluate our approach `ddnnife` and `query-ddnnf` as d-DNNF reasoners. Furthermore, we found that the latest version of `d4` provides an option to compute a set of queries given a CNF, which we include in our evaluation as `d4-query`. To the best of our knowledge, this option is also not mentioned in any publication. We evaluate each solver and combination of compiler & reasoner on 62 industrial feature models collected from previous empirical evaluations [118, 121, 161, 171]. For each feature model, we evaluate the tools for computing the configuration count for (1) every feature and (2) for 200 random partial configurations with different numbers of included/excluded features.

4. https://www.cril.univ-artois.fr/kc/d-DNNF-reasoner.html

Results Figure 44A shows the cumulative runtimes of the #SAT solvers and the d-DNNF-based solver combinations for computing the cardinality of features and partial configurations. For the d-DNNF-based measurements, the runtimes include both compilation and reasoning time. Note that the results of `sharpSAT` and `dSharp` are omitted as both produced inconsistent results for various queries, while the remaining tools always computed the same model counts for a given query. Overall, 8 out of the 62 could not be compiled to d-DNNF and also not evaluated by any #SAT solver. `ddnnife` combined with `d4` successfully evaluated both computations (i.e., cardinality of all features and cardinality of 200 random configurations) for all remaining 54 models, while the #SAT solvers together only succeeded for 47 feature models. On the feature model with the largest performance difference, `ddnnife` with `d4` is ~ 15,388 times faster than the fastest #SAT solver. This is a reduction by 99.99%, essentially decreasing runtimes from several days to seconds. Overall, `ddnnife` with `d4` is the fastest strategy requiring 10.8 minutes to evaluate all cardinalities for the 54 feature models followed by `d4-query` with 46.9 minutes. No other strategy successfully evaluated all 54 remaining feature models.

Observation – RO3 (Accelerate the Multitude of Queries)

Reusing d-DNNFs can reduce runtimes for single analyses from days to few seconds.

Observation – RO3 (Accelerate the Multitude of Queries)

Combining our proposal `ddnnife` with `d4` appears to be the most promising strategy for the cardinality of features $|\mathbb{CS}_f|$ and partial configurations $|\mathbb{CS}_C|$.

Limitations We refer to our original work, for a thorough discussion of potential threats to validity [C2]. One considerable limitation is the reliance

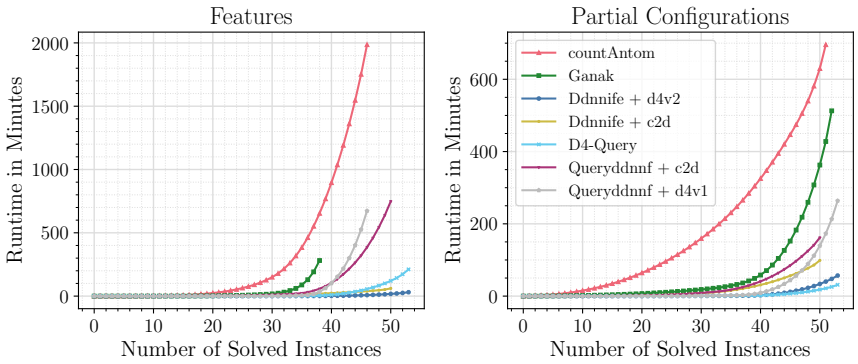


Figure 44a: *Runtime Comparisons for Computing Cardinality of Features and Cardinality of Partial Configurations (Adapted from [C2], licensed under CC-BY 4.0, https://creativecommons.org/licenses/by/4.0/)*

on synthesized partial configurations due to the lack of publicly available queries. As partial configurations in practice would often differ depending on (1) the application, (2) the subject system, and (3) the concrete instance (e.g., user decisions), it is hard to predict real-world instances. Our goal was to cover various potential scenarios by generating random configurations of varying sizes. Still, examining the benefits of reusing d-DNNFs for real-world partial configurations would be valuable.

5.4 Projected d-DNNF Compilation

In this section, we discuss the contributions of our **ASE'24** publication [C3] on *projected* d-DNNF compilation. A *projection* is essentially an adaptation of a formula containing only a subset of the variables but preserving all transitive dependencies [9]. We propose projected d-DNNF compilation to derive d-DNNFs which only contain features of interest.

5.4.1 Targeted Problem

With our approach of projected d-DNNF compilation, we tackle the scalability of feature-model analyses relying on a subset of the features [94, 96, A1]. For configuration counting, our approach directly covers the application *variability of feature subsets* (cf. Section 3.2). Using the produced d-DNNF, we can also perform further analyses tractable on the d-DNNF. For instance, in the context of configuration counting, we can perform analyses depending on the cardinality of feature models, features, or partial configurations on that projected d-DNNF using the proposal from our **TOSEM'24** publication [C2].

Relevance Often, only parts of the configuration knowledge are relevant to a certain stakeholder [2, 94, 96]. While a mechanic may focus on feature interactions between hardware components, a software tester may discard all non-software features when conducting software-in-the-loop tests [61]. Keeping track of the configurability including non-relevant features (e.g., hardware interactions) may induce considerable overhead [94].

Automated analysis on projected feature models is computationally expensive [94, 96, 126]. The complexity stems from the fact that we cannot just discard the features we are not interested in, but need to preserve transitive dependencies [2, 96, 126]. Otherwise, the projected feature model may induce valid configurations that are not valid for the original feature model. With the current state of the art, the feature model is first projected using *slicing* [126] via *logical resolution*, which is computationally demanding, especially when removing many variables [126]. Then, the projected feature model is used for subsequent analysis, which also often depends on computationally expensive operations [A2C1], such as SAT or #SAT.

Application in Practice Within an industrial collaboration [94, 96], we tackled the mismatch between the variability in the problem space (i.e., valid configurations) and solution space (i.e., actual products). To this end, we introduced feature models spanning over both user-selectable features and implementation artifacts. We then examined the solution space variability by employing *variability of a feature subset* (cf. Table 35A). There, we used a combination of slicing [2, 126] and black-box #SAT solving [215]. However, we regularly faced scalability issues for more complex feature models. By introducing *projected d-DNNF compilation*, we were able to analyze feature models within seconds that previously ran for hours without any result. Furthermore, while all #SAT solvers and d-DNNF compilers failed for a feature model representing the Linux kernel [B1, A2C1], we can compile d-DNNFs for many parts of it (i.e., feature subsets) with our approach [C3].

Limitations of Related Work As far as we know, projected knowledge compilation to *any* target language has not been considered yet. In the product-line domain, the de facto standard is subsequently applying feature-model slicing [2, 126] and then performing the required analysis directly [94, 96]. However, slicing [126] and typically also the subsequent analysis (e.g., counting) are computationally expensive and, thus, this approach may induce scalability issues. In the model counting domain, several projected model counters have been proposed [9, 78, 198], but are limited to single-shot invocations. As discussed in Chapter 2, d-DNNFs are typically derived from the trace of an exhaustive DPLL traversal [54, 133, 152]. Projected model counters also rely on exhaustive DPLL traversals [9, 198] and, thus, could in theory be extended for projected d-DNNF compilation. However, optimizations commonly used in projected model counting often only preserve equi-countability which is not suitable for d-DNNF compilation as queries on a d-DNNF only satisfying equi-countability may produce faulty results.

5.4.2 Concept of Projected d-DNNF Compilation

Figure 46A visualizes the idea of our approach. With projected d-DNNF compilation, we merge two expensive operations, namely slicing and d-DNNF compilation, into one step. Similar to regular d-DNNF compilation, we perform an exhaustive DPLL traversal to derive the d-DNNF. However, during the traversal, we discard the variables that should not be part of the projection. While slicing and knowledge compilation algorithms have existed before this thesis [2, 126, 133, 165], this is the first proposal for *projected* knowledge compilation. Our work [C3] focuses on the instantiation of this novel idea on d-DNNFs.

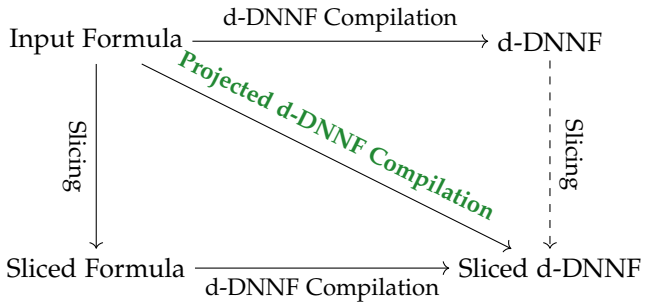


Figure 46A: *Commuting Diagram Showcasing Projected d-DNNF Compilation (Adapted from [C3], licensed under CC-BY 4.0, https://creativecommons.org/licenses/by/4.0/)*

Figure 46B shows an adaptation of the example exhaustive DPLL traversal shown in Figure 12A, where we create a projection containing the variables A, B and C. In the basic algorithm, there are two major differences between projected and regular exhaustive DPLL. First, we can only branch on projected variables. Second, whenever the remaining subformula consists of only non-projected variables, we check if the subformula is satisfiable and stop the exhaustive descent for this branch. Using the resulting trace, we can derive a d-DNNF that includes only projected variables but still considers all transitive dependencies.

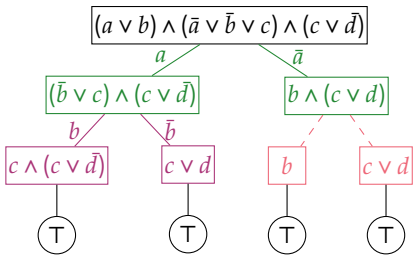


Figure 46B: *Projecting c and d in Exhaustive DPLL Traversal shown in Figure 12A*

Optimizations We thoroughly describe the optimizations and required adaptations in the **ASE'24** publication [C3]. Here, we provide only a short overview. Before starting compilation, we preprocess the formula with the following strategies: *equivalent literal substitution* [135], *partial resolution*, and *backbone propagation* [110]. To identify disconnected components [18] (cf. Chapter 2), we use *weighted hypergraph partitioning* [185], which is an adaptation of regular hypergraph partitioning as used in `d4` to incorporate projections. During solving, we use an adaptation of the commonly employed variable ordering heuristic called *variable-state-aware decaying sum (VSADS)* [183]. Furthermore, we always eliminate variables that are not part of the projection that are pure [111] (i.e., for literal l there is no negated literal $\bar{l}$ in the formula).

Tool Support We implemented our approach of projected d-DNNF compilation as an extension of `d4`, the best performing d-DNNF compiler for configuration counting [B1, A2C1, C2]. We call this extension `pd4`.[5] Projected d-DNNF compilation is available as additional mode of `d4` next to, e.g., model counting or regular d-DNNF compilation. All described

5. https://github.com/SoftVarE-Group/d4v2

optimizations are also included in pd4. Our reasoner ddnnife (cf. Section 5.3) comes with pd4 as integrated d-DNNF compiler to also allow CNFs with flags indicating variables to project.

5.4.3 Evaluation

Setup To compare pd4 to the state of the art, we consider two baselines. First, we use a combination of slicing [126] and regular d-DNNF compilation as baseline to derive projected d-DNNFs. Krieter et al. [126] compare the performance of multiple slicing algorithms. As slicing baseline, we use the most efficient slicing algorithm identified by Krieter et al. [126], which is implemented in FeatureIDE [148]. Furthermore, we consider the most efficient d-DNNF compiler for feature-model analysis d4 [133, B1, A2C1]. Second, we compare pd4 to projected model counters. While those solvers only provide the projected model count and no d-DNNF, the underlying strategies are closely related to our techniques. Here, we use the best performing projected model counters from the model counting competition 2023.[6] Regarding subject systems, we evaluated each tool on three datasets. First, we had access to *industrial* feature models with projections applied in practice from our industry partner. Second, we used feature models that are also *industrial but with synthesized projections* as no actual projections were publicly available. Third, we compared the approaches on the instances from the projected model counting track from the *model counting competition* 2023, which represent formulas from various problem domains.

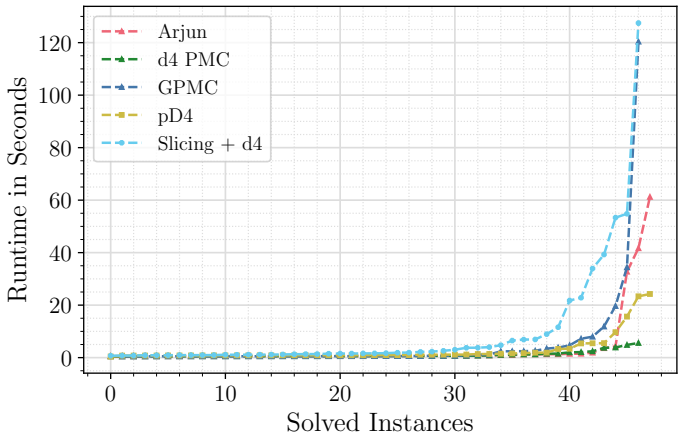
6. When we submitted the original paper, those were the most recent available results.

Results Figure 48A shows the runtimes of each considered tool on the three datasets. For every dataset, pd4 scales to more instances than subsequently using slicing and regular d-DNNF compilation. On the industrial projection, pd4 requires 136 seconds for 47 (out of 48) projections, while slicing followed by regular compilation requires 454 seconds for 46 projections. On the synthesized projections on industrial feature models, pd4 successfully evaluated 4,516 (out of 4,900) projections, while slicing followed by d4 only scaled to 4,170 projections. Here, it is important to notice that many of the industrial feature models are rather small and induce short runtimes for all evaluated tools. For instance, the 3,000 easiest instances were evaluated within one second each by *every tool*. On the competition instances, pd4 solves 159 (out of 200) while slicing followed by d4 solves only 52 instances.

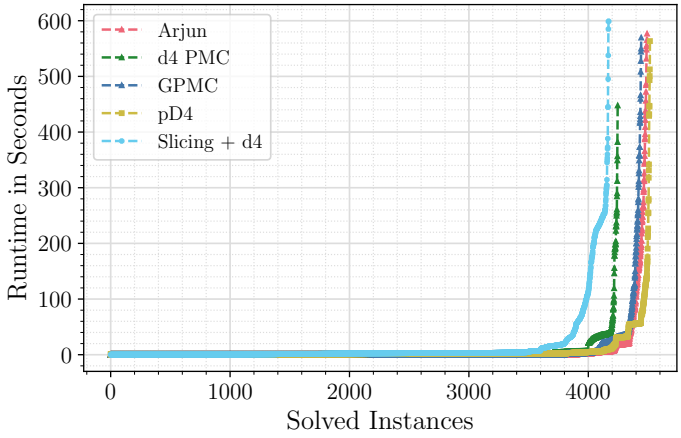
Observation – RO3 (Accelerate the Multitude of Queries)

Our approach pd4 performs substantially better than existing solutions for deriving a projected d-DNNF.

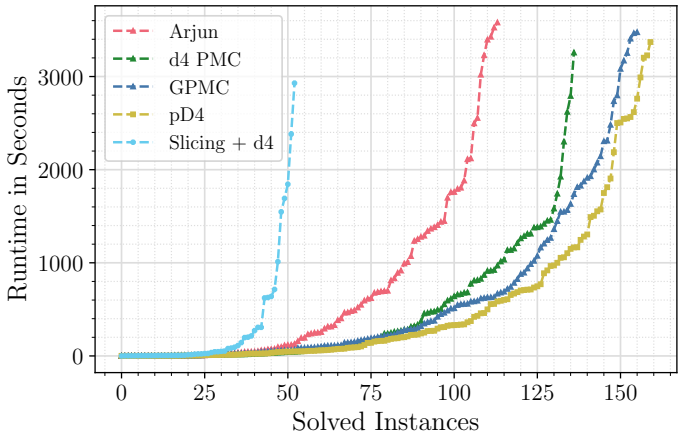
Our proposal pd4 is also faster than each projected model counter on every dataset. On the industrial projections, the respectively fastest projected model counter also solved 47 instances but required more runtime (136 vs 176 seconds). For industrial models with synthesized projections (4,516 vs 4,492 solved instances) and competition instances (159 vs 155 solved instances), pd4 scaled to the highest number of instances.



(a) *Industrial Feature Models & Industrial Projections*



(b) *Industrial Feature Models & Random Projections*



(c) *Model Counting Competition Instances*

Figure 48a: *Performance Projected d-DNNF Compilation (Adapted from [C3], licensed under CC-BY 4.0, https://creativecommons.org/licenses/by/4.0/)*

Observation – RO3 (Accelerate the Multitude of Queries)

Our projected d-DNNF compiler scales to more instances than every projected model counter.

An interesting observation of our empirical evaluation is that compiling a projected d-DNNF can scale where regular d-DNNF compilation fails. For instance, the Linuxv2.6.33 feature model could not be evaluated by regular d-DNNF compilers, even with a timeout of 24 hours [B1] (cf. Section 4.2). Using our approach for projected d-DNNF compilation, we were able to successfully compile a d-DNNF for 96 out of 100 projections on the Linux feature model.

Observation – RO3 (Accelerate the Multitude of Queries)

In many cases, we can compute projected d-DNNFs for feature models where deriving a regular d-DNNF does not scale.

Limitations We discuss potential threats to the validity of our empirical evaluation in the original work [C3]. One considerable limitation of our results is the lack of real-world projections for most feature models. While we had access to projections applied in practice from our industry partner, these instances are limited to one domain and even one company. Overall, the strong results of our approach over all datasets suggest that `pd4` is a promising candidate for analyses based on projections in general. Nevertheless, identifying more real-world projections and examining the performance of the different tools would be valuable.

5.5 Pseudo-Boolean d-DNNF Compilation

In this section, we discuss the contributions of this thesis on *pseudo-Boolean* d-DNNF compilation [C4]. Pseudo-Boolean logic supports numeric constants but *variables* are still limited to Boolean (i.e., {0,1}). In particular, a pseudo-Boolean constraint follows the structure $\sum_{i=1}^{n} a_i \cdot v_i \geq b$ [41]. Here, a_i and b are numeric constants and v_i is a Boolean variable that maps to {0, 1}. Each pseudo-Boolean formula is a conjunction over such constraints.

5.5.1 Targeted Problem

With our approach of pseudo-Boolean d-DNNF compilation [C4], we tackle the lack of scalable reasoning for more expressive feature-modeling constructs. Various common feature-modeling constructs [21], such as cardinality [51], attribute constraints [25, 26], or typed features [25], are ill-suited for translation into conjunctive normal form (CNF) [34, 154]. As available strategies for model counting are generally limited to CNFs as input [40, 78, 182, 194, 215], the scalability of applying configuration counting for feature models containing those constructs is questionable. By using pseudo-Boolean logic, we aim to strike a balance between smaller encodings for additional constructs compared to CNF, but still facilitate scalable reasoning. Here, the advantage of pseudo-Boolean formulas is that the flexibility of the constraints allow more efficient encoding while we still benefit from the performance advantages of only managing Boolean

variables. Our particular selection of targeted feature-modeling constructs is based on the language level mechanism of the Universal Variability Language (UVL) [25]. UVL is a commonly used specification for feature models that is developed as a community effort [70, 72, 84, 144, 180, 181, 205]. We contributed to the language design, analyses, and tool ecosystem for UVL with several publications [25, 144, 181, 205, 207, 210] (cf. Section A). For managing different feature-modeling constructs, UVL has an extensible design that categorizes the constructs into the levels *Boolean*, *arithmetic*, and *feature types* [25, 210]. With our approach, we target the expressive constructs group cardinality, feature cardinality, and attribute constraints which overall covers the *Boolean* and the *arithmetic* level of UVL. Nevertheless, there are still relevant constructs which we do not target here, such as typed features [25], that are also not well suited for pseudo-Boolean logic. With our approach of pseudo-Boolean d-DNNF compilation, we can compile feature models encoded as pseudo-Boolean formulas into *Boolean* d-DNNFs, which let us reuse the d-DNNF for repetitive queries with available algorithms and tools [193, C2], such as presented in Section 5.3.

Relevance Most variability languages include expressive constructs [21]. In a survey on available variability languages, Beek et al. [21] identified 13 languages. 11 out of those 13 languages include expressive constructs that are not well suited for a pure Boolean representation [21]. UVL, which has been widely employed [70, 73, 105, 144, 154, 180, 181, 205, 207] since its initial release in 2021 [203], also includes various of those constructs [25]. Many of UVL's design decisions, including the addition of those expressive constructs, are the result of a survey evaluating the preferences of researchers and practitioners [203]. Overall, there are many indicators that variability modeling community considers these constructs relevant. However, previously proposed solutions for configuration counting [36, 76, 136], including the proposals in this thesis [B1, A2C1, C3, C2], do not support these constructs.

Application in Practice Variability modeling tools commonly used in industry also use the targeted expressive constructs [28, 32, 62, 180]. Berger et al. [28] performed a literature survey on industrial practices for variability modeling. The six most popular (according to their results [28]) public tools, namely pure::variants [180], GEARS [32], FeatureIDE[7], DOPLER [62], SAP [90], and XFeature,[8] all use such expressive concepts. Furthermore, several case studies observed that these constructs are mandated in practice [27, 34, 107, 166, 224]. In particular, all feature-modeling constructs targeted within our work [C4], namely group cardinality, feature cardinality, and attribute constraints, are used in real-world projects [27, 34, 166, 224]. Our approach is motivated also by our industrial collaboration where we faced scalability issues with CNF-based solvers on constraints over feature attributes (i.e., non-functional properties attached to features).

7. Note that the FeatureIDE format only supports basic feature models, but FeatureIDE also supports UVL [205].

8. `https://www.pnp-software.com/XFeature/`

Limitations of Related Work The vast majority of reasoning engines that can be employed for configuration counting are based on propositional logic [40, 76, 78, 99, 129, 133, 134, 194, 215]. This limitation to propositional logic also includes our other contributions [B1, C3, C2, B2]. Both typically considered strategies for model counting, namely solvers based

on exhaustive DPLL and knowledge compilation to d-DNNF, are typically limited to CNFs as input [40, 54, 133, 152, 182, 215]. The model counting competition is also limited to tracks featuring propositional formulas in CNF [78]. While the competition includes a weighted model counting track, these variable weights are not used in the constraints.

The applicability of solvers that support more expressive constructs is limited. Satisfiability Modulo Theories (SMT) support various expressive constructs [57, 59] and have been applied in the product-line domain [151, 201]. However, SMT solving is limited to checking satisfiability [57, 201], finding optimal solutions [57, 201], and to *approximate* counting [45]. To the best of our knowledge, exact model counting or knowledge compilation has not been considered for SMT. Benavides et al. [26] employ constraint programming for counting the number of valid configurations of feature models with attribute constraints, but their approach scaled for feature models with only up-to 25 features. Yang and Meel [227] also recently proposed a knowledge compiler for pseudo-Boolean logic. However, their proposal is based on algebraic decision diagrams [13], which are tractable for more analyses but also more complex (w.r.t. compilation effort) compared to d-DNNFs. Preliminary experiments also revealed that (1) their tool often produces inconsistent results compared to `d4` and our proposal `p2d` and (2) our proposal performs substantially better.

5.5.2 Approach

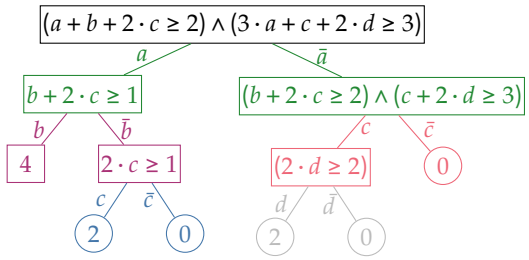
Pseudo-Boolean Feature Model Encoding There are various feature modeling constructs considered in the literature [21]. In our **TSE** submission [C4], we limited ourselves to constructs supported in UVL due to its wide usage [70, 144, 154, 180, 181, 205, 210], broad coverage of feature modeling constructs desired in research and practice [21, 25, 28, 203]. Overall, we target the following constructs: group cardinality, feature cardinality, and attribute constraints. The selection covers all of UVLs language features except typed features. While typed features have been considered at times in feature modeling [132, 166], they also substantially increase the complexity of automated reasoning [59, 144, 154].

A pseudo-Boolean formula is a generalization of CNFs, as a clause $(v_1 \vee \ldots \vee v_n)$ can be represented as $\sum_{i=1}^{n} 1 \cdot v_i \geq 1$. This added flexibility of pseudo-Boolean formulas over CNFs can be used for smaller representations of feature-modeling constructs. For example, an alternative group (i.e., select exactly one feature) requires a quadratic number of clauses or adding artificial variables in CNF [117], but can be represented as $p + \sum_{i=1}^{n} -1 \cdot c_i = 0$ in pseudo-Boolean logic.[9] Here, p is the parent and c_i are the child features in the alternative. Numeric attributes can also be straightforwardly represented in pseudo-Boolean logic. Following a common definition of attributes [25], which is also used in UVL, an attribute is a constant value that is activated or not depending on the selection status of the feature. For instance, a feature f_i may have a price of $p_i = 10$. If a configuration includes f_i the price is set to 10 otherwise it is set to 0. Consequentially, an attribute can be simply specified as a summand (i.e., $p_i \cdot f_i$). Note that in some interpretations [19, 158], feature attributes can be configured separately from the feature and, thus, essentially behave like typed features. We do not target this interpretation with our proposal. For the concrete encodings of the remaining feature-modeling constructs, we refer to the **TSE** submission [C4].

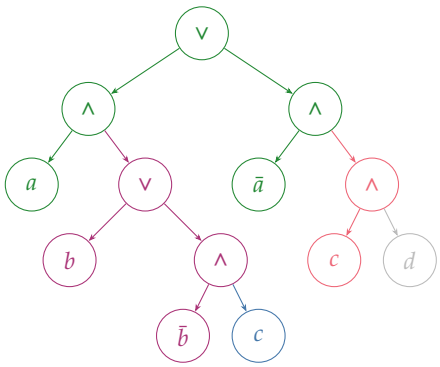
9. Note that the equality can be straightforwardly normalized to match the structure $\sum_{i=1}^{n} a_i \cdot v_i \geq b$ [C4].

Pseudo-Boolean d-DNNF Compilation To realize pseudo-Boolean d-DNNF compilation, we employ and adapt successful techniques from pure Boolean solving. As basic algorithm, we derive the d-DNNF from an exhaustive DPLL traversal [33]. Figure 52A shows an example for our approach of pseudo-Boolean d-DNNF compilation. While similar on an abstract level, exhaustive DPLL and most optimizations heavily rely on the structure of the input formula, which is typically CNF and, thus, require considerable adaptation on a more detailed level. The major difference regarding the basic algorithm (i.e., exhaustive DPLL algorithm) between Boolean and pseudo-Boolean is the processing of decision variables in the constraints. Processing decisions d in CNF clauses is straightforward: if there is a literal d, the clause is satisfied and can be removed. If there is a negated literal $\bar{d}$, this literal $\bar{d}$ is removed from the clause. If there is an empty clause, the formula is falsified. For pseudo-Boolean constraints, we need to update both sides of the inequality accordingly. Furthermore, detecting whether the constraint is satisfied or falsified for all remaining assignments requires further processing.

The optimizations inspired from CNF-based solvers also require considerable adaptation. We refer to our **TSE** submission [C4] for details on all employed optimizations and adaptations required to apply those for pseudo-Boolean d-DNNF compilation. For instance, Boolean constraint propagation (BCP) [215] revolves around unit clauses for solving on CNFs as the respective decisions are directly implied. In contrast, it is not trivial to determine which literals are implied by a pseudo-Boolean constraint. We use an adaptation proposed for pseudo-Boolean SAT solving [41] where we check if the largest constant a_i is required for the inequality in $\sum_{i=1}^{n} a_i \cdot v_i \geq b$ to be satisfied. If so, we can propagate the decision v_i.



(a) *Exhaustive DPLL Traversal for Pseudo-Boolean Formula*



(b) *d-DNNF Derived from Pseudo-Boolean Trace*

Figure 52A: *Pseudo-Boolean d-DNNF Compilation*

Tool Support For both steps of our approach, namely encoding and compilation, we provide publicly available tooling. The encoder UVL2PB[10] takes UVL models as input and translates them into a pseudo-Boolean formula in opb[11] format. We provide the encoding in Java to match the UVL parser ecosystem [210]. Our compiler p2d[12] takes opb files as input and compiles a d-DNNF in the d4 [133] format. Alternatively, p2d can be used to directly compute the model count.

10. https://github.com/TUBS-ISF/pseudo-boolean-uvl-encoder

11. opb is the standard format used for representing pseudo-Boolean formulas.

12. https://github.com/TUBS-ISF/p2d

5.5.3 Evaluation

With our empirical evaluation, we aim to compare pseudo-Boolean d-DNNF compilation with the CNF-based state of the art for compiling feature models. Hereby, we focus on both required steps, namely *encoding* and *compilation*, of both approaches. For each step, we compare the approaches regarding required runtime and artifact size.

Methodology As far as we know, model counting or d-DNNF compilation have not been applied for the considered expressive feature-modeling constructs. To still enable CNF-based reasoning for the targeted constructs as baseline, we use (1) UVL conversion strategies to translate expressive constructs to basic constructs, (2) CNF translations from FeatureIDE [148], and (3) the best performing d-DNNF compiler for feature models d4 [133]. For the pseudo-Boolean pipeline, we use our proposals UVL2PB and p2d. Even though the variability modeling tools most often employed in industrial practice [28] support such constraints, there is a lack of *publicly* available feature models with such constructs. To provide as indicative insights as possible on the behavior for feature models with such constructs, we evaluate the approaches on the following datasets: (1) publicly available industrial feature models with basic constructs, (2) expressive constructs in isolation with various changing parameters, (3) randomly generated models with different combinations of constructs, and (4) confidential feature models with expressive constructs from our industry partner. For more details on the methodology regarding subject systems and experiment execution, we refer to the original work [C4].

Results Table 54A provides an overview of the performance results of both approaches. Translating the feature model into a pseudo-Boolean formula is significantly faster for every dataset including the industrial basic feature models compared to translating into CNF. There are considerable differences for all expressive constructs. For instance, for group cardinalities in isolation, we can encode all group cardinalities (largest group had 5,000 features) with pseudo-Boolean logic, but could translate only up to 13 features to CNF. The Boolean encoding hit the timeout for both confidential models from our industry partner, while the pseudo-Boolean encoding required less than two seconds for each.

> 🔎 **Observation – RO3 (Accelerate the Multitude of Queries)**
>
> Pseudo-Boolean encoding yields performance benefits over all considered feature-modeling constructs.

Even though our work focuses on expressive constructs, our approach appears to be competitive with the CNF-based approach over basic feature models. For the industrial feature models from literature [B3], 11

Table 54a: *Performance Comparison for Boolean and Pseudo-Boolean Pipeline (Published in [C4] under CC-BY 4.0, https://creativecommons.org/licenses/by/4.0/)*

Dataset	Enc. (%)		Enc. Size		Comp. (%)		d-DNNF Size	
	CNF	PB	CNF	PB	d4	p2d	d4	p2d
Real-World Basic	97.4	97.4	5,073	**3.987**	**81.6**	80.3	**2,310**	2,892
Isolated								
Alternative	98.8	**100**	1.6M	**4,002**	12.5	**53.8**	2,751	2,751
Group Cardinality	7.05	**100**	42	**22**	7.05	**87.4**	44	44
Feature Cardinality	79.1	**99.0**	42K	**2,486**	70.7	**76.8**	26k	**1,800**
Addition	9.15	**100**	37	**26**	9.15	**90.5**	**40.5**	59.5
Multiplication	60.9	**94.7**	**32**	986	60.9	**76.2**	**39**	308
Division	48.1	**100**	**22**	135	48.1	**100**	**20**	46
Synthesized								
Just Group Card.	81.7	**100**	9,709	**8,667**	81.7	**100**	614	**613**
Just Feature Card.	97.7	**100**	41K	**21K**	96.7	96.7	1,257	**970**
Just Expression	100	100	9,860	**9,191**	100	100	**686**	795
Just Aggregate	3.67	**100**	9,411	**7,979**	3.67	**100**	489	489
All but Group Card.	2.0	**100**	5,202	**3,983**	2.0	**97.3**	151	**150**
All but Feature Card.	4.33	**100**	2,140	**1,964**	4.33	**99.3**	125	**113**
All but Expression	8.33	**99.7**	88K	**35K**	8.33	**98.3**	1,313	**872**
All but Aggregate	81.7	**100**	34K	**18K**	81.0	**97.7**	1,072	**973**
All constructs	6.0	**100**	11K	**5,384**	6.0	**96.3**	276	**186**
Industrial	0	**100**	N.A.	**14K**	0	0	N.A.	N.A.

features models could be evaluated by neither approach, p2d failed for two additional ones and d4 for one additional one. p2d is faster for 32 feature models and d4 for 31 feature models. While d4 overall requires less runtime (11.8 vs 15.9 minutes), the differences over the entire dataset are not statistically significant.

Observation – RO3 (Accelerate the Multitude of Queries)

Pseudo-Boolean d-DNNF compilation is competitive for basic feature models.

The pseudo-Boolean pipeline scales to a higher number of features for all expressive constructs in isolation. For *group cardinality* (5,000 vs 15 features in largest successfully evaluated model), *alternative groups* (4,300 vs 1,000), *feature cardinality* (5,000 vs 3,600), and *addition* (2,100 vs 19) there are substantial differences between the pseudo-Boolean and CNF-based pipelines. For the remaining constructs, namely *multiplication* (76.2% vs 60.9%) and *division* (100% vs 48.1%), p2d also scaled to considerably more instances. On the synthesized models, p2d with UVL2PB failed for only 1.6% while d4 with CNF translation failed for 57.4%.

Observation – RO3 (Accelerate the Multitude of Queries)

Pseudo-Boolean d-DNNF compilation yields substantial runtime improvements for expressive constructs over CNF into d-DNNF compilation.

Limitations For a thorough discussion of threats to the validity of our evaluation, we refer to the original work [C4]. In this thesis, we discuss two major limitations to consider for the transferability of our results. First, our evaluation includes only two industrial feature models with expressive constructs. While we put considerable effort in assembling datasets that provide indicators on performance in practice for those constructs, it remains unclear if those translate to industrial feature models with those constructs. Nevertheless, our approach outperformed the CNF-based approach substantially over instances including the expressive constructs in various combinations and numerous parameterizations. Second, we limited ourselves to one Boolean and one pseudo-Boolean encoding for every expressive construct. Switching the encodings may have considerable impact on the performances. While, to the best of our knowledge, there are no alternative suggestions for the pseudo-Boolean encoding, this is probably due to a research gap. There may be more efficient encodings, at least for some constructs (e.g., multiplication and division). For translation to CNF, there is a plethora of encodings available. In particular, we found various encodings for cardinality constraints [14, 34, 196]. Furthermore, bit-blasting could be employed for numerical attributes [154]. The main problem for those CNF encodings is that it is unclear whether these encodings produce correct[13] results or are more efficient. Also considering the plethora of different encodings would increase the complexity of an already broad empirical evaluation. Hence, we consider an examination of more encodings as valuable future work but out of scope for this thesis.

13. For the vast majority of cardinality encodings we saw, only equisatisfiability [14, 34, 196] is preserved which is not sufficient for configuration counting [131].

5.6 Summary

In this chapter, we present our contributions that aim to accelerate analyses based on configuration counting [A2C1, C3, C2]. Our proposed strategies revolve around knowledge compilation to the deterministic decomposable negation normal form (d-DNNF). We focus on d-DNNFs as they appear to be a sweet spot between enabled feature-model analyses and efficiency for compilation (cf. Section 5.2). Furthermore, after counting once with modern algorithms [182, 215], we already have the required knowledge for building the d-DNNF [54, 133, 152] as d-DNNFs can be derived from a #SAT trace [54, 133, 152]. We realize optimized queries on d-DNNFs for configuration counting (cf. Section 5.3) which cover all applications relying on repetitive queries targeting the cardinality of feature and the cardinality of partial configurations. After showcasing the benefits of having a d-DNNF, we propose two novel compilation strategies. First, we propose projected d-DNNF compilation to accelerate configuration counting on a projection (e.g., only software features) of the feature model (cf. Section 5.4). Second, we facilitate d-DNNF compilation of expressive feature-modeling constructs, such as cardinalities or attribute constraints with pseudo-Boolean d-DNNF compilation (cf. Section 5.5). By combining the contributions presented in this chapter, we tackle the scalability of all applications presented in Chapter 3.

Across our contributions, the empirical evaluations indicate that our proposals can substantially improve performance over the state of the art. Note that the previous state of the art consists of solutions already specialized to counting which have been optimized for decades [54, 77, 119, 133, 182, 194, 215]. For analyses depending on multiple counting queries (e.g., on all features), we can reduce runtimes from days to seconds by

reusing d-DNNFs [C2]. Our projected d-DNNF compiler pd4 is able to compile more instances to d-DNNFs with only features of interest on every considered dataset than previous approaches [C3]. Furthermore, we can often compute projected d-DNNFs for systems (e.g., the Linux kernel [29, 132]), for which we cannot compile a regular d-DNNF. With pseudo-Boolean d-DNNF compilation, we can compile substantially larger instances of expressive constructs while being competitive on basic feature models [C4].

The contributions presented in this chapter could also be valuable for reasoning outside of configuration counting. Each proposed strategy operates on logical formulas as input and, thus, could be applied to other problem domains that employ solving such operations on logical formulas. For instance, model counting is used in program reliability [212], encryption [20], or verifying neural networks [15]. Projections on formulas or projected model counting have been, e.g., employed for analysis of program flows [116] and analyzing networks in molecular biology [85]. Pseudo-Boolean reasoning is used in other domains (e.g., economics [92]) and the targeted expressive constructs, such as cardinality constraints, have been employed in further domains [14, 139, 196]. Our empirical evaluations already provide some indicators on the benefits in other domains. With our projected d-DNNF compiler pd4, we outperformed the state of the art on the projected model counting track of the model counting competition 2023 which covers a large variety of instance domains [C3]. Our pseudo-Boolean d-DNNF compiler p2d showed a strong performance compared to CNF-based tools on expressive constructs in isolation which do not represent feature models [C4]. Notably, our research on reusing d-DNNFs has already been applied to other domains than feature models [199].

In addition to the conceptual and empirical results, we provide a tool ecosystem that facilitates efficient configuration counting. For each proposed strategy, we provide publicly available tooling. Starting from a feature model, we can derive a formula in CNF via FeatureIDE [148] or in pseudo-Boolean format via our proposal UVL2PB [C4]. Then, we can perform model counting or compile to d-DNNF using (1) available tools for regular compilation/counting [133, 215], (2) pd4 [C3] for projections (i.e., analyzing a subset of features), or (3) p2d [C4] for pseudo-Boolean formulas. The compiled d-DNNFs can then be used in ddnnife [C2], enabling a plethora of feature model analyses based on configuration counting, but also other problem settings, such as satisfiability, enumeration [138], or deriving optimal configurations [179]. The tooling for every listed contribution has already been successfully applied within industry collaborations. In particular, reusing d-DNNFs with ddnnife [C2] and projected d-DNNF compilation with pd4 [C3] has substantially improved the scalability of analyses in practice.

6 Conclusion & Future Work

Configuration counting refers to the computation of the number of valid configurations induced by a product line. Even though configuration counting has been considered for various product-line engineering tasks in research and practice [44, 47, 76, 96, 97, 98, 129, 146, 161], its applicability is still hindered by several obstacles. Knowledge about possible applications is *limited and scattered* across numerous publications [44, 47, 76, 96, 97, 98, 129, 146, 161]. Furthermore, computation instructions are often missing or imprecise and, thus, realizing an identified application requires considerable expertise. Another obstacle is the *scalability* of configuration counting. Configuration counting relies on computationally complex operations that may exceed acceptable runtimes with existing solutions. Knowledge about the *performance of available solutions* is limited. Furthermore, as revealed by our work [B1], there are no *scalable solutions* for various problem settings in configuration counting [A2C1, A1]. Particularly, we found no solution scaling to the numerous repeated counting queries required for many applications. In this thesis, we unleash the potential of configuration counting by tackling these obstacles with several contributions.

Our Advances in Configuration Counting

We tackle the obstacle of *limited and scattered knowledge* on configuration counting with a survey of applications [A2C1, A1] (cf. Chapter 3). The survey presents applications that span the whole product-line development lifecycle and are relevant for various stakeholders. For each application, we provide computation instructions that can be reduced to four different categories of configuration counting [A1]. Even though we do not consider our list of applications exhaustive of the possibilities facilitated by configuration counting, our survey indicates the potential benefits and relevance of configuration counting. Furthermore, our contributions simplify getting an overview on potential use cases and how to compute results for them.

With a large-scale empirical evaluation (cf. Chapter 4), we inspect the *performance of available solutions* [B1, B2]. Examining off-the-shelf solutions, we found several indicators [103, 136, 174] that #SAT solving is the most promising available strategy for configuration counting compared to enumerating solutions [82, 148, 174] and ad-hoc approaches focusing on feature trees [75, 76, 100]. We gathered 19 #SAT solvers and evaluated them on industrial feature models [B1] to examine their performance for configuration counting. Here, we found that the best-performing #SAT solvers are capable of analyzing most industrial feature models. Still, we identified scalability issues as (1) two systems could not be analyzed by any solver and (2) applications often rely on numerous counting operations [A2C1] which may quickly exceed acceptable runtimes for many systems. Both issues may even worsen over time as we found indicators

that systems become harder for #SAT solvers during evolution [132, B3, B2].

To *improve the scalability* of the numerous counting operations required for configuration counting, we provide several contributions that substantially improve performance over the state of the art [A2C1, C3, C2] (cf. Chapter 5). Our overall strategy to tackle the scalability issues relies on knowledge compilation [56]. Here, the feature model is translated to a target language with beneficial properties for the efficiency of followup operations (e.g., counting). Surveying and examining available target languages, we identified the deterministic decomposable negation normal form (d-DNNF) as a sweet spot for configuration counting [A2C1]. Then, we further showcase the advantages of d-DNNFs for configuration counting by providing optimized algorithms to query the d-DNNF [C2]. Following the clear indicators on the benefits of *having* a d-DNNF, we propose novel strategies for *compiling* feature models into d-DNNF for different problem settings. First, we propose projected d-DNNF compilation as the first instance of projected knowledge compilation to derive d-DNNFs representing the variability of a feature subset [C3]. Second, we introduce pseudo-Boolean d-DNNF compilation [C4] to tackle configuration counting for expressive feature-modeling constructs that are ill-suited for state-of-the-art solvers. For each proposed strategy, we provide open-source tooling. Our empirical evaluations strongly indicate that our contributions substantially improve the scalability for all tackled problem settings compared to the previous state of the art [C3, C2, C4]. For each problem setting, our proposals enable scalable analysis of more systems and lead to substantial reductions in runtime (e.g., from days to seconds in some cases) compared to state-of-the-art counting strategies that have been optimized for decades [54, 78, 119, 182, 194].

Our contributions tackling the scalability [A2C1, C3, C2, C4] are not only motivated by applications from literature, but also from our industry collaborations. In particular, every contribution successfully tackled a scalability issue that occurred in relevant analyses performed on production systems. We already employed each developed tool in practice within our industry collaborations. Our contributions helped to enable hundreds of analyses a day, which were not feasible before this thesis.

Overall, we advance the state of the art for configuration counting in several dimensions. First, we showcase the potential of configuration counting and present concrete computation instructions by providing an overview of applications, including several novel ideas we employed to tackle issues with variability in industrial practice [A2C1, A1]. Second, we examine the performance of state-of-the-art solutions for configuration counting with a large scale empirical evaluation revealing the capabilities of #SAT solving, but also limitations in its scalability [B3, B1, B2]. Third, with four contributions revolving around knowledge compilation to d-DNNF, we tackle the scalability of configuration counting for different practically relevant applications. For each tackled application, our approaches substantially reduce runtimes compared to the previous state of the art [C3, C2, C4]. Within our industry collaborations, we could already observe a substantial impact of our contributions regarding the applicability of configuration counting in practice.

Future Work

Even though this thesis tackled various issues in the context of configuration counting, we identified several aspects that could benefit from further research.

Understanding Complex Instances The question of *why certain systems are too complex for configuration counting* still requires further examination [130, 228]. While we provide some insights on correlations between structural metrics and scalability of configuration counting [B1, B2], those cannot fully explain which properties of systems like the Linux kernel cause the major issues in scalability for configuration counting. Being able to predict whether a feature model can be analyzed could help avoid computations which are bound to fail and, thus, save resources for those demanding computations. However, deriving properties that are capable of universally identifying instances that may not scale is a complex problem and actionable insights are still limited, also for model counting in general [77, 78, 87]. A major problem is that the hardness of instances heavily relies on heuristics employed in the reasoning engines [78, 119, 182, 183, 215]. This dependency on heuristics can also be used for tweaking performance. It may be valuable to better understand the problems with complex feature models to tailor heuristics.

Scalability for Complex Instances In our empirical evaluations [B1, C3, C2], several systems could not be fully analyzed with configuration counting. Even though off-the-shelf #SAT solvers are continuously improving [78, 119, 194], configurable systems also grow in complexity over time [132, B3, B2]. We made some advances toward enabling configuration counting for overly complex systems. With projected d-DNNF compilation [C3], we can perform configuration counting on such complex systems for a subset of features in many cases. Nevertheless, it may be valuable and potentially necessary to develop solutions tailored to solving complex feature models. One approach may be exploiting alternative groups (i.e., select exactly one feature from that group) as those occur regularly in feature models [B3] and heuristics based on alternatives are successfully applied in OBDD compilation [64, 102]. Another approach may be parametrizing off-the-shelf tools. The performance of #SAT solvers heavily relies on the heuristics and optimizations employed. Deriving promising parameters for configuration counting or even for specific systems may improve runtimes. Such hyperparameter tuning was out of scope for this thesis, but several advised student theses give rise to optimism [17, 65, 66, 186].

Evolution of Feature Models In Chapter 4, we examined the evolution of a few systems regarding the computational effort for #SAT and some structural properties (e.g., number of features) [132, B3, B2]. We found a clear trend that the feature models corresponding to those specific systems get more complex over time; the number of features and constraints increases and the scalability of #SAT solving decreases [132, B3, B2]. Future research on understanding and possibly remedying the effects of the growing complexity may be valuable. In particular, examining the questions *do our observations translate to other systems?*, *what causes the increase in complexity?*, or *do we need countermeasures to the ever-increasing complexity?* The potential

need for countermeasures to tackle complexity has already been observed independently of configuration counting [5, 8]. Configuration counting can also be employed to effectively tackle complexity issues as deliberate *variability reduction* can be facilitated using configuration counting [A1] (cf. Section 3.2). The statement *"you can't control what you can't measure"* [60] appears to apply to effectively reducing the variability of a feature model. Without automated support, it is infeasible to grasp the impact of a change (e.g., adapting constraints) on the variability [24]. Within our industry collaborations, we already employed configuration counting to successfully reduce complexity and its growth over time. Before introducing configuration counting, the number of valid configurations grew exponentially over time [B2]. We also observed this exponential growth in other projects [118, 156, 171, B3]. With the help of deliberately controlling the variability via configuration counting, the number of valid configurations kept a similar level over years for multiple projects in our industry collaboration.

Non-Boolean Configuration Counting With pseudo-Boolean d-DNNF compilation [C4], we improved the scalability for configuration counting on various feature-modeling constructs. Still, the solutions in this thesis focus on *Boolean* features, which each have only a binary selection status; included or excluded in a configuration. While research on automated analysis in general focuses on Boolean features, many variability languages and systems in practice consider also other types of features. Common examples are multi-state [132, 166], numeric [25, 166], or string features [25, 166]. For feature models containing such non-Boolean feature types, our configuration counting proposals cannot be directly applied. One strategy could be using conversions to Boolean logic [132, 154] (e.g., with bit-blasting [154]). Further research on the scalability of such techniques or alternative proposals may be valuable to make configuration counting applicable in non-Boolean settings.

Study Usage in Practice This thesis focuses on pushing the capabilities of configuration counting. We present applications [A2C1, A1] and provide scalable solutions for many scenarios [B1, A2C1, C2, B2]. We have not targeted how to effectively present application results to potential users of configuration counting. While configuration counting has been applied in practice [96, 129], including in our industry collaborations, there have been no studies on how results of configuration counting are used in practice. Future research examining questions like *how to effectively present results of configuration counting?*, *which applications are actually used in practice?*, or *which exact counting operations are frequently performed?* may be valuable to expand the application of configuration counting in practice.

Extension to Solution Space In product-line engineering, features and their dependencies are often referred to as *problem space* [96]. The *solution space* refers to implementation artifacts that are used for generating the actual products. These artifacts often come with *presence conditions* that decide whether the artifact is included for a given configuration. The core contributions in this thesis focus on the problem space [B3, B1, A2C1, C3, A1, C2, B2, C4]. In other work [94, 96], we facilitated analysis on the solution space to some extent by integrating presence conditions and artifacts in the feature model and then apply slicing [2, 126]. Using these integrated

feature models, we can apply the applications presented in Chapter 3 and our solution proposals (cf. Chapter 5) to reason about the solution space. In particular, we applied projected d-DNNF compilation [C3] to successfully tackle the scalability issues we observed with the original slicing-based solution [94, 96]. However, for many applications, the semantics when applied to the solution space are not clear. In previous work [96], we discussed the semantics of only four applications when adapted for the solution space (from overall 21 in Chapter 3). Furthermore, extending configuration counting for analysis of the solution space may facilitate new applications. For instance, the reliability examination of programs as performed by Teuber and Weigl [212] could be extended for software product lines. When additional or adapted applications for the solution space are developed, our reasoning proposals from Chapter 5 should still be valuable as our contributions cover a broad range of computational problem settings and even more counting queries could be required.

Applying Our Proposals Outside of Configuration Counting Each of our solution proposals presented in Chapter 5 operates on logical formulas. While those formulas mostly represent product lines in our empirical evaluations, the ideas and even the optimizations could also be useful for instances from other problem domains (i.e., in general for knowledge compilation and model counting). In our empirical evaluation on projected d-DNNF compilation [C3], we outperformed the state of the art for projected model counting even on instances not related to product lines. The isolated concepts we evaluated with pseudo-Boolean d-DNNF compilation, such as cardinality constraints, are also used in other domains [14, 196]. Hence, the performance benefits may transfer to those domains. Reusing d-DNNFs with our proposal `ddnnife` [C2] has been applied for schedule planning by Speck et al. [199]. Overall, as initial observations suggest the applicability of our approaches to other problem domains, future research examining their exact benefits could be valuable.

Appendix

A Related Work of the Author

In addition to the publications corresponding to the core contributions of this thesis, the author was involved in various other research contributions during his PhD studies. In this section, we discuss the relation of those other publications to this thesis.

A.1 Additional Publications Corresponding to Core Contributions

For three core contributions, we published work that directly corresponds to the respective main publication [A2C1, C2, B2]. For our work on reusing d-DNNFs [C2], we provided a technical report that also included an ablation study on the integrated optimizations [209]. We also published two extended abstracts [206, 208] corresponding to talks featuring our core contributions on evaluating #SAT solvers [B1] and examining knowledge compilation target languages [A2C1].

[209] Chico Sundermann, Heiko Raab, Tobias Heß, Thomas Thüm, and Ina Schaefer. *Exploiting d-DNNFs for Repetitive Counting Queries on Feature Models*. Tech. rep. arXiv:2303.12383. Cornell University Library, 2023. DOI: 10.48550/arXiv.2303.12383

[208] Chico Sundermann, Elias Kuiter, Tobias Heß, Heiko Raab, Sebastian Krieter, and Thomas Thüm. "On the Benefits of Knowledge Compilation for Feature-Model Analyses". In: *Proc. Int'l Systems and Software Product Line Conf. (SPLC)*. ACM, 2024, p. 217. DOI: 10.1145/3646548.3676540

[206] Chico Sundermann, Tobias Heß, Michael Nieke, Paul Maximilian Bittner, Jeffrey M. Young, Thomas Thüm, and Ina Schaefer. "Evaluating State-of-the-Art #SAT Solvers on Industrial Configuration Spaces". In: *Proc. Software Engineering (SE)*. Gesellschaft für Informatik, 2024, pp. 67–68. DOI: 10.18420/sw2024_18

A.2 Knowledge Compilation to Binary Decision Diagrams

In our thesis, knowledge compilation plays a central role for improving the scalability of configuration counting. While we derived d-DNNFs as a sweet spot for configuration counting due to the supported poly-time queries and the computational effort to compile, other knowledge compilation languages can be relevant for other problem settings [A2C1]. Binary decision diagrams (BDDs) are considerably harder to compile than d-DNNFs, but enable a richer set of poly-time queries [A2C1]. For instance, BDDs support poly-time conjunction which could be helpful for some instances of the cardinality of a formula [A2C1]. In the following research contributions, as part of the PhD studies of Heß, we examined the scalability of binary decision diagrams for feature modeling together with Tobias Heß [101, 102, 103].

[103] Tobias Heß, Chico Sundermann, and Thomas Thüm. "On the Scalability of Building Binary Decision Diagrams for Current Feature Models". In: *Proc. Int'l Systems and Software Product Line Conf. (SPLC)*. ACM, 2021, pp. 131–135. DOI: 10.1145/3461001.3474452

[102] Tobias Heß, Sean Niklas Semmler, Chico Sundermann, Jacobo Torán, and Thomas Thüm. "Towards Deterministic Compilation of Binary Decision Diagrams From Feature Models". In: *Proc. Int'l Systems and Software Product Line Conf. (SPLC)*. ACM, 2024, pp. 136–147. DOI: 10.1145/3646548.3672598

[101] Tobias Heß, Tobias Müller, Chico Sundermann, and Thomas Thüm. "ddueruem: A Wrapper for Feature-Model Analysis Tools". In: *Proc. Int'l Systems and Software Product Line Conf. (SPLC)*. ACM, 2022, pp. 54–57. DOI: 10.1145/3503229.3547032

A.3 Feature-Model Analysis

A wide variety of feature-model analyses have been proposed in the literature [24, 95, 97]. Besides our core contributions to configuration counting, we also contributed to related analyses. In previous work [94, 96] within the PhD studies of Hentze, we extended feature models with presence conditions (i.e., constraints that control whether an implementation artifact is part of the product) to analyze the solution space of a product line. Here, we used a combination of slicing and subsequent analyses, namely sampling [94] and counting [96]. We tackled the encountered scalability issues within our contribution of projected d-DNNF compilation [C3].

[96] Marc Hentze, Chico Sundermann, Thomas Thüm, and Ina Schaefer. "Quantifying the Variability Mismatch Between Problem and Solution Space". In: *Proc. Int'l Conf. on Model Driven Engineering Languages and Systems (MODELS)*. IEEE, 2022, pp. 322–333. DOI: 10.1145/3550355.3552411

[94] Marc Hentze, Tobias Pett, Chico Sundermann, Sebastian Krieter, Thomas Thüm, and Ina Schaefer. "Generic Solution-Space Sampling for Multi-Domain Product Lines". In: *Proc. Int'l Conf. on Generative Programming: Concepts & Experiences (GPCE)*. ACM, 2022. DOI: 10.1145/3564719.3568695

In our thesis, we repeatedly examined our approaches for the Linux kernel, which is one of the most complex systems we encountered. In other work within PhD studies of Kuiter, we examined the Linux kernel and its complexity for SAT and #SAT during evolution more thoroughly [132].

[132] Elias Kuiter, Chico Sundermann, Thomas Thüm, Tobias Heß, Sebastian Krieter, and Gunter Saake. "How Configurable is the Linux Kernel? Analyzing Two Decades of Feature-Model History". In: *Trans. on Software Engineering and Methodology (TOSEM)* (2025). DOI: 10.1145/3729423

The translation of feature models into CNFs is essential for most of our approaches as they rely on CNF as input [C3, C2]. Furthermore, the off-the-shelf solutions we examined also rely on CNF [B1, A2C1]. There are various properties to consider when selecting a suitable CNF transformation. In other work, we examined and evaluated such properties for distilling a taxonomy of different transformations [131]. An important observation regarding this thesis is that not all transformations preserve the model count.

[131] Elias Kuiter, Sebastian Krieter, Chico Sundermann, Thomas Thüm, and Gunter Saake. "Tseitin or not Tseitin? The Impact of CNF Transformations on Feature-Model Analyses". In: *Proc. Int'l Conf. on Automated Software Engineering (ASE)*. ACM, 2022, 110:1–110:13. DOI: 10.1145/3551349.3556938

Another important observation of this thesis is that the performance of an analysis depends on a variety of factors, such as the underlying problem (e.g., SAT or #SAT), the number of queries of that problem, and the input instance [B1, A2C1, C3, C2, C4]. We build on those observations and discuss the importance of considering these aspects in future research in a *new and controversial ideas* paper [130], which we supported as co-author.

[130] Elias Kuiter, Tobias Heß, Chico Sundermann, Sebastian Krieter, Thomas Thüm, and Gunter Saake. "How Easy Is SAT-Based Analysis of a Feature Model?" In: *Proc. Int'l Working Conf. on Variability Modelling of Software-Intensive Systems (VaMoS)*. ACM, 2024, pp. 149–151. DOI: 10.1145/3634713.3634733

In further work we supported as co-author [124, 201], we present and examine solutions for tackling feature-model analyses beyond counting. First, we employed Satisfiability Modulo Theories (SMT) [58] for various feature-model analyses including fault explanations and deriving optimized configurations [201]. Second, we incrementally adapted modal implication graphs, a data structure that can be used to accelerate decision propagation [124].

[201] Joshua Sprey, Chico Sundermann, Sebastian Krieter, Michael Nieke, Jacopo Mauro, Thomas Thüm, and Ina Schaefer. "SMT-Based Variability Analyses in FeatureIDE". in: *Proc. Int'l Working Conf. on Variability Modelling of Software-Intensive Systems (VaMoS)*. ACM, 2020. DOI: 10.1145/3377024.3377036

[124] Sebastian Krieter, Rahel Arens, Michael Nieke, Chico Sundermann, Tobias Heß, Thomas Thüm, and Christoph Seidl. "Incremental Construction of Modal Implication Graphs for Evolving Feature Models". In: *Proc. Int'l Systems and Software Product Line Conf. (SPLC)*. ACM, 2021, pp. 64–74. DOI: 10.1145/3461001.3471148

A.4 Variability Modeling with UVL

The Universal Variability Language (UVL) is a textual format for specifying feature models [25]. UVL is developed as a community effort within the MODEVAR initiative [22]. In 2021 [203], we proposed an initial version of UVL that has been extended [25, 210] and widely adopted in feature modeling already [70, 72, 84, 104, 140, 144, 180, 181, B3, 205]. This wide adoption of UVL streamlines research and tool development. Previously, a considerably larger variety of formats were used by different research groups, which hindered exchange [3, 21, 46, 113, 148, 149]. In our thesis, we benefit from the wide adoption in our collection of industrial feature models [B3] as each model is provided in UVL. Furthermore, with pseudo-Boolean d-DNNF compilation, we can focus on constructs from UVL [C4]. We contributed to the language design of UVL with the following publications [25, 203, 210].

[203] Chico Sundermann, Kevin Feichtinger, Dominik Engelhardt, Rick Rabiser, and Thomas Thüm. "Yet Another Textual Variability Language? A Community Effort Towards a Unified Language". In:

Proc. Int'l Systems and Software Product Line Conf. (SPLC). ACM, 2021, pp. 136–147. DOI: 10.1145/3461001.3471145

[210] Chico Sundermann, Stefan Vill, Thomas Thüm, Kevin Feichtinger, Prankur Agarwal, Rick Rabiser, José A. Galindo, and David Benavides. "UVLParser: Extending UVL With Language Levels and Conversion Strategies". In: *Proc. Int'l Systems and Software Product Line Conf. (SPLC)*. ACM, 2023, pp. 39–42. DOI: 10.1145/3579028.3609013

[25] David Benavides, Chico Sundermann, Kevin Feichtinger, José A. Galindo, Rick Rabiser, and Thomas Thüm. "UVL: Feature Modelling With the Universal Variability Language". In: *J. Systems and Software (JSS)* 225 (2025). DOI: 10.1016/j.jss.2024.112326

Another important aspect for the usability of a language is available tool support. Across various publications, we developed concepts and tool support for (1) modeling with UVL [144, 205], (2) analysis [144], (3) exchange [181], (4) transformation [73], and (5) generation [207] of UVL models. We used the transformation to derive the UVL models for our collection [B3] and the generator to create synthetic UVL models for our evaluation of pseudo-Boolean d-DNNF compilation [C4].

[205] Chico Sundermann, Tobias Heß, Dominik Engelhardt, Rahel Arens, Johannes Herschel, Kevin Jedelhauser, Benedikt Jutz, Sebastian Krieter, and Ina Schaefer. "Integration of UVL in FeatureIDE". in: *Proc. Int'l Workshop on Languages for Modelling Variability (MODEVAR)*. ACM, 2021, pp. 73–79. DOI: 10.1145/3461002.3473940

[181] David Romero-Organvídez, José A Galindo, Chico Sundermann, José-Miguel Horcas, and David Benavides. "UVLHub: A Feature Model Data Repository Using UVL and Open Science Principles". In: *J. Systems and Software (JSS)* 215 (2024). DOI: 10.1016/j.jss.2024.112150

[144] Jacob Loth, Chico Sundermann, Tobias Schrull, Thilo Brugger, Felix Rieg, and Thomas Thüm. "UVLS: A Language Server Protocol for UVL". in: *Proc. Int'l Systems and Software Product Line Conf. (SPLC)*. ACM, 2023, pp. 43–46. DOI: 10.1145/3579028.3609014

[73] Kevin Feichtinger, Chico Sundermann, Thomas Thüm, and Rick Rabiser. "It's Your Loss: Classifying Information Loss During Variability Model Roundtrip Transformations". In: *Proc. Int'l Systems and Software Product Line Conf. (SPLC)*. ACM, 2022, pp. 67–78. DOI: 10.1145/3546932.3546990

[207] Chico Sundermann, Tobias Heß, Rahel Sundermann, Elias Kuiter, Sebastian Krieter, and Thomas Thüm. "Generating Feature Models with UVL's Full Expressiveness". In: *Proc. Int'l Workshop on Languages for Modelling Variability (MODEVAR)*. ACM, 2024, pp. 61–65. DOI: 10.1145/3646548.3676602

We also conducted three tutorials on UVL for which we published respective extended abstracts [106, 125, 204].

[204] Chico Sundermann, Kevin Feichtinger, José A. Galindo, David Benavides, Rick Rabiser, Sebastian Krieter, and Thomas Thüm. "Tutorial on the Universal Variability Language". In: *Proc. Int'l Systems and Software Product Line Conf. (SPLC)*. ACM, 2022, 260:1. DOI: 10.1145/3546932.3547024

[125] Sebastian Krieter, Kevin Feichtinger, José A. Galindo, David Benavides, Rick Rabiser, Chico Sundermann, and Thomas Thüm. "Second Tutorial on the Universal Variability Language". In: *Proc. Int'l Systems and Software Product Line Conf. (SPLC)*. ACM, 2023, p. 273. DOI: 10.1145/3579027.3609002

[106] José Miguel Horcas, Chico Sundermann, Malte Grave, José A. Galindo, Thomas Thüm, Rick Rabiser, and David Benavides. "Third Tutorial on the Universal Variability Language: Practical Extensions and Tool Ecosystem". In: *Proc. Int'l Systems and Software Product Line Conf. (SPLC)*. New York, NY, USA: ACM, 2025, p. 241. ISBN: 9798400720246. DOI: 10.1145/3744915.3749149

B Advised Theses

In this section, we shortly discuss theses we advised and their relation to this thesis.

Several core contributions build upon a thesis we advised. In particular, advised students were involved in reusing d-DNNFs [176, C2], deriving a feature-model collection [38, B3], projected d-DNNF compilation [143, C3], and pseudo-Boolean d-DNNF compilation [C4, 222].

[143] Jacob Loth. "Projected d-DNNF Compilation for Feature Models". Master's Thesis. University of Ulm, 2023

[176] Heiko Raab. "Exploiting d-DNNFs for Efficient Cardinality-Based Feature-Model Analyses". Bachelor's Thesis. University of Ulm, 2022. DOI: 10.18725/OPARU-43414

[38] Vincenzo Brancaccio. "A Systematic Literature Review Towards a Representative Feature-Model Benchmark". Master's Thesis. University of Ulm, 2022. DOI: 10.18725/OPARU-46759

[222] Stefan Vill. "Pseudo Boolean d-DNNF Compilation for Analyzing Expressive Feature-Model Formats". Master's Thesis. University of Ulm, 2025. DOI: 10.18725/OPARU-56664

Two advised theses target reusing information for subsequent counting queries when editing feature models. With incremental #SAT [184], the idea is to maintain information collected by the solver (e.g., variable orderings or conflict clauses) between calls on adaptations of the same original feature model. To incorporate changes on an existing d-DNNF [177], we aim to find parts of the d-DNNF that needs to be updated after a change and only perform a partial recompilation. Both approaches are applicable also for the cardinality of a formula (cf. Section 3.3).

[184] Dominik Schiessl. "An Incremental #SAT Solver for Efficient Analysis of Feature Models". Master's Thesis. University of Ulm, 2022

[177] Heiko Raab. "Incrementally Adapting d-DNNFs to Cope with Feature-Model Evolution". Master's Thesis. University of Ulm, 2023. DOI: 10.18725/OPARU-52288

Within two further advised theses, we used machine learning to tailor parameters of approximate #SAT solvers for histories of feature models [186] and for meta-solving (i.e., selecting the most promising solver from a portfolio for a given feature model) [65].

[186] Clemens Schmid. "Parameterizations for Approximate #SAT Solvers". Bachelor's Thesis. University of Ulm, 2023

[65] Raphael Dunkel. "One Solver to Rule All Feature Models – Or Not? Addressing the Algorithm Selection Problem for #SAT". Bachelor's Thesis. University of Ulm, 2024. DOI: 10.18725/OPARU-52711

In another advised thesis, we preprocessed the input feature model with various strategies to simplify subsequent counting queries [93].

[93] Yvonne Heimowski. "Simplifying Feature Models for Better Scalability of #SAT Solvers". Bachelor's Thesis. University of Ulm, 2022

To extend the solving capabilities for our d-DNNF reasoner `ddnnife`, we developed multiple sampling approaches on d-DNNFs in two advised theses [138, 179].

[138] Lars Licha. "Cutting Edge T-Wise Sampling With ddnnife". Bachelor's Thesis. University of Ulm, 2023

[179] Jonathan Riesland. "Optimizing T-Wise Sampling on d-DNNFs with Objective Functions". To appear. Bachelor's Thesis. University of Ulm, 2024

Another advised thesis build the base for the extensible language design of the Universal Variability Language (UVL) [221].

[221] Stefan Vill. "Language Levels for the Universal Variability Language: An Extension Mechanism and Conversion Strategies". Bachelor's Thesis. University of Ulm, 2022. DOI: 10.18725/OPARU-47708

In this thesis, we discussed the deliberate reduction of variability to decrease complexity on several occasions. However, identifying effective changes to reduce the variability is not trivial. One suggestion we made is using configuration counting. In a thesis we advised, we tried to gather more information on effective changes by identifying variability drivers (i.e., features that induce a lot of variability) [164].

[164] Lukas Ostheimer. "Identification of Variance Driving Features in Feature Models". Bachelor's Thesis. University of Ulm, 2022

C Student's Theses of the Author

In this section, we discuss the relation of the student's theses written by the author to this thesis. In the bachelor's thesis [200], we examined the usage of SMT solvers for optimizing configurations based on feature attributes. In Section 5.5, we also target feature attributes amongst other feature-modeling constructs. However, while we consider constraints over feature attributes in this thesis, we have not considered them in our bachelor's thesis which vastly simplifies automated reasoning.

[200] Joshua Sprey and Chico Sundermann. "Computing Attribute Ranges for Partial Configurations With JavaSMT". Bachelor's Thesis. TU Braunschweig, 2018. DOI: 10.24355/dbbs.084-201807060926-0

In our master's thesis [202], we already targeted configuration counting and conducted some initial exploration on a few contributions of this PhD thesis. First, provided an initial list of applications including computation constructions, which we later extended and made several improvements regarding the classification and the formalization of computation instructions in our **VaMoS'21** publication [A1]. Second, we performed an initial empirical evaluation of #SAT solvers on industrial evaluation. In our **EMSE'23** publication [B1], we considered seven additional #SAT solvers, examined also the structure of feature models, and conducted more profound statistical analysis. Third, we proposed a naive algorithm to traverse d-DNNFs for configuration counting. However, without optimizations, traversing d-DNNFs essentially re-traverses the entire #SAT trace. Consequentially, the d-DNNF algorithms presented in the master's thesis only yielded small runtime benefits. The optimizations which we later introduced in our **TOSEM'24** publication [C2] reduced the runtimes by multiple orders of magnitudes in many cases, scaling substantially better than the initial proposal in our master's thesis and also than the state of the art. Overall, within the master's thesis, we explored several ideas for tackling issues in configuration counting on a superficial level. With this PhD thesis, we extended on promising contributions of our master's theses [B1, A1, C2] and proposed several novel ideas [B3, A2C1, C3, C4].

[202] Chico Sundermann. "Applications of #SAT Solvers on Product Lines". Master's Thesis. TU Braunschweig, 2020. DOI: 10.24355/dbbs.084-202009161329-0

Bibliography

[1] Iago Abal, Jean Melo, Stefan Stănciulescu, Claus Brabrand, Márcio Ribeiro, and Andrzej Wąsowski. "Variability Bugs in Highly Configurable Systems: A Qualitative Analysis". In: *Trans. on Software Engineering and Methodology (TOSEM)* 26.3 (2018), 10:1–10:34. DOI: 10.1145/3149119. — On page 1.

[2] Mathieu Acher, Philippe Collet, Philippe Lahire, and Robert B. France. "Slicing Feature Models". In: *Proc. Int'l Conf. on Automated Software Engineering (ASE)*. IEEE, 2011, pp. 424–427. DOI: 10.1109/ASE.2011.6100089. — On pages 3, 4, 44, 45, 60.

[3] Mathieu Acher, Philippe Collet, Philippe Lahire, and Robert B. France. "Familiar: A Domain-Specific Language for Large Scale Management of Feature Models". In: *Science of Computer Programming (SCP)* 78.6 (2013), pp. 657–681. — On pages 7, 65.

[4] Mathieu Acher, Patrick Heymans, Philippe Collet, Clément Quinton, Philippe Lahire, and Philippe Merle. "Feature Model Differences". In: *Proc. Int'l Conf. on Advanced Information Systems Engineering (CAiSE)*. Springer, 2012, pp. 629–645. DOI: 10.1007/978-3-642-31095-9_41. — On pages 36, 37.

[5] Mathieu Acher, Luc Lesoil, Georges Aaron Randrianaina, Xhevahire Tërnava, and Olivier Zendra. "A Call for Removing Variability". In: *Proc. Int'l Working Conf. on Variability Modelling of Software-Intensive Systems (VaMoS)*. ACM, 2023, pp. 82–84. DOI: 10.1145/3571788.3571801. — On page 60.

[6] Dimitris Achlioptas, Zayd S Hammoudeh, and Panos Theodoropoulos. "Fast Sampling of Perfectly Uniform Satisfying Assignments". In: *Proc. Int'l Conf. on Theory and Applications of Satisfiability Testing (SAT)*. Springer, 2018, pp. 135–147. — On page 18.

[7] Carlos Ansótegui, María Luisa Bonet, and Jordi Levy. "On the Structure of Industrial SAT Instances". In: *Proc. Int'l Conf. on Principles and Practice of Constraint Programming (CP)*. Springer, 2009, pp. 127–141. — On page 27.

[8] Vard Antinyan. "Revealing the Complexity of Automotive Software". In: *Proc. Europ. Software Engineering Conf./Foundations of Software Engineering (ESEC/FSE)*. ACM, 2020, pp. 1525–1528. DOI: 10.1145/3368089.3417038. — On page 60.

[9] Rehan Abdul Aziz, Geoffrey Chu, Christian Muise, and Peter Stuckey. "#∃ SAT: Projected Model Counting". In: *Proc. Int'l Conf. on Theory and Applications of Satisfiability Testing (SAT)*. Springer, 2015, pp. 121–137. — On pages 4, 38, 44, 45.

[10] Fahiem Bacchus, Shannon Dalmao, and Toniann Pitassi. "DPLL With Caching: A New Algorithm for #SAT and Bayesian Inference". In: *Electronic Colloquium in Computational Complexity*. 2003. — On page 27.

[11] Ebrahim Bagheri and Dragan Gasevic. "Assessing the Maintainability of Software Product Line Feature Models Using Structural Metrics". In: *Software Quality Journal (SQJ)* 19.3 (2011), pp. 579–612. On pages 16, 17.

[12] Ebrahim Bagheri, Tommaso Di Noia, Dragan Gasevic, and Azzurra Ragone. "Formalizing Interactive Staged Feature Model Configuration". In: *J. Software: Evolution and Process* 24.4 (2012), pp. 375–400. On pages 18, 19, 23.

[13] R. Iris Bahar, Erica A. Frohm, Charles M. Gaona, Gary D. Hachtel, Enrico Macii, Abelardo Pardo, and Fabio Somenzi. "Algebraic Decision Diagrams and Their Applications". In: *Proc. Int'l Conf. on Computer-Aided Design (ICCAD)*. IEEE, 1993, pp. 188–191. On page 51.

[14] Olivier Bailleux and Yacine Boufkhad. "Efficient CNF Encoding of Boolean Cardinality Constraints". In: *Proc. Int'l Conf. on Principles and Practice of Constraint Programming (CP)*. Berlin, Heidelberg: Springer, 2003, pp. 108–122. ISBN: 978-3-540-45193-8. On pages 55, 56, 61.

[15] Teodora Baluta, Shiqi Shen, Shweta Shinde, Kuldeep S. Meel, and Prateek Saxena. "Quantitative Verification of Neural Networks and Its Security Applications". In: *Proc. SIGSAC Conf. on Computer and Communications Security (CCS)*. London, United Kingdom: ACM, 2019, pp. 1249–1264. ISBN: 9781450367479. DOI: 10.1145/3319535.3354245. On pages 11, 56.

[16] Don Batory. "Feature Models, Grammars, and Propositional Formulas". In: *Proc. Int'l Systems and Software Product Line Conf. (SPLC)*. Springer, 2005, pp. 7–20. DOI: 10.1007/11554844_3. On pages 3, 7, 9, 21, 25.

[17] Jan Baudisch. "Tailoring Hypergraph Partitioning for Efficient d-DNNF Compilation". To appear. Bachelor's Thesis. University of Ulm, 2025. On page 59.

[18] Roberto J. Bayardo Jr. and Joseph Daniel Pehoushek. "Counting Models Using Connected Components". In: *Proc. Conf. on Artificial Intelligence (AAAI)*. AAAI Press, 2000, pp. 157–162. On pages 12, 46.

[19] Guillaume Bécan, Razieh Behjati, Arnaud Gotlieb, and Mathieu Acher. "Synthesis of Attributed Feature Models from Product Descriptions". In: *Proc. Int'l Systems and Software Product Line Conf. (SPLC)*. Nashville, Tennessee: ACM, 2015, pp. 1–10. ISBN: 9781450336130. DOI: 10.1145/2791060.2791068. On page 51.

[20] Gabrielle Beck, Maximilian Zinkus, and Matthew Green. "Automating the Development of Chosen Ciphertext Attacks". In: *Proc. USENIX Security Symposium (USS)*. USENIX Association, 2020, pp. 1821–1837. ISBN: 978-1-939133-17-5. On pages 11, 56.

[21] Maurice H. ter Beek, Klaus Schmid, and Holger Eichelberger. "Textual Variability Modeling Languages: An Overview and Considerations". In: *Proc. Int'l Workshop on Languages for Modelling Variability (MODEVAR)*. ACM, 2019, pp. 151–157. DOI: 10.1145/3307630.3342398. On pages 7, 21, 49–51, 65.

[22] David Benavides, Rick Rabiser, Don Batory, and Mathieu Acher. "First International Workshop on Languages for Modelling Variability (MODEVAR 2019)". In: *Proc. Int'l Systems and Software Product Line Conf. (SPLC)*. 2019, pp. 323–323. On page 65.

[23] David Benavides, Antonio Ruiz-Cortés, and Pablo Trinidad. "Automated Reasoning on Feature Models". In: *Proc. Int'l Conf. on Advanced Information Systems Engineering (CAiSE)*. 2005, pp. 491–503. On pages 17, 18.

[24] David Benavides, Sergio Segura, and Antonio Ruiz-Cortés. "Automated Analysis of Feature Models 20 Years Later: A Literature Review". In: *Information Systems* 35.6 (2010), pp. 615–708. DOI: 10.1016/J.IS.2010.01.001. On pages 1–3, 8, 9, 16–18, 21, 25, 36, 60, 64.

[25] David Benavides, Chico Sundermann, Kevin Feichtinger, José A. Galindo, Rick Rabiser, and Thomas Thüm. "UVL: Feature Modelling With the Universal Variability Language". In: *J. Systems and Software (JSS)* 225 (2025). DOI: 10.1016/j.jss.2024.112326. On pages 7, 30, 49–51, 60, 65, 66.

[26] David Benavides, Pablo Trinidad, and Antonio Ruiz-Cortés. "Using Constraint Programming to Reason on Feature Models". In: *Proc. Int'l Conf. on Software Engineering and Knowledge Engineering (SEKE)*. 2005, pp. 677–682. On pages 23, 26, 49, 51.

[27] Thorsten Berger, Divya Nair, Ralf Rublack, Joanne M. Atlee, Krzysztof Czarnecki, and Andrzej Wąsowski. "Three Cases of Feature-Based Variability Modeling in Industry". In: *Proc. Int'l Conf. on Model Driven Engineering Languages and Systems (MODELS)*. Springer, 2014, pp. 302–319. DOI: 10.1007/978-3-319-11653-2_19. On pages 41, 50.

[28] Thorsten Berger, Ralf Rublack, Divya Nair, Joanne M. Atlee, Martin Becker, Krzysztof Czarnecki, and Andrzej Wąsowski. "A Survey of Variability Modeling in Industrial Practice". In: *Proc. Int'l Workshop on Variability Modelling of Software-Intensive Systems (VaMoS)*. ACM, 2013, 7:1–7:8. DOI: 10.1145/2430502.2430513. On pages 3, 7, 25, 26, 36, 41, 50, 51, 53.

[29] Thorsten Berger, Steven She, Rafael Lotufo, Andrzej Wąsowski, and Krzysztof Czarnecki. "A Study of Variability Models and Languages in the Systems Software Domain". In: *IEEE Trans. on Software Engineering (TSE)* 39.12 (2013), pp. 1611–1640. DOI: 10.1109/TSE.2013.34. On pages 7, 8, 26, 27, 30, 56.

[30] Armin Biere. "PicoSAT Essentials". In: *J. Satisfiability, Boolean Modeling and Computation* 4 (2008), pp. 75–97. On page 11.

[31] Armin Biere, Marijn Heule, Hans van Maaren, and Tony Walsh. *Handbook of Satisfiability*. IOS Press, 2009. On pages 2, 3, 11, 12, 22, 25, 26.

[32] Big Lever Software Inc. *Gears: A Software Product Line Engineering Tool*. Website: http://www.biglever.com/solution/product.html. Accessed: 2010-11-04. 2010. On page 50.

[33] Elazar Birnbaum and Eliezer L Lozinskii. "The Good Old Davis-Putnam Procedure Helps Counting Models". In: *J. Artificial Intelligence Research (JAIR)* 10 (1999), pp. 457–477. On pages 11, 12, 52.

[34] Paul Maximilian Bittner, Thomas Thüm, and Ina Schaefer. "SAT Encodings of the At-Most-k Constraint – A Case Study on Configuring University Courses". In: *Proc. Int'l Conf. on Software Engineering and Formal Methods (SEFM)*. Ed. by Peter Csaba Ölveczky and Gwen Salaün. Springer, 2019, pp. 127–144. DOI: 10.1007/978-3-030-30446-1_7. On pages 49, 50, 55.

[35] Barry Boehm, A Winsor Brown, Ray Madachy, and Ye Yang. "A Software Product Line Life Cycle Cost Estimation Model". In: *Proc. Int'l Symposium on Empirical Software Engineering (ISESE)*. IEEE, 2004, pp. 156–164. — On page 16.

[36] Pierre Bourhis, Laurence Duchien, Jérémie Dusart, Emmanuel Lonca, Pierre Marquis, and Clément Quinton. *Reasoning on Feature Models: Compilation-Based vs. Direct Approaches*. Tech. rep. Cornell University Library, 2023. DOI: 10.48550/arXiv.2302.06867. — On pages 13, 16, 17, 36, 37, 50.

[37] Thomas F. Bowen, Frank S. Dworack, Ching-Hua Chow, Nancy Griffeth, Gary E. Herman, and Yow-Jian Lin. "The Feature Interaction Problem in Telecommunications Systems". In: *Proc. Int'l Conf. on Software Engineering for Telecommunication Switching Systems (SETSS)*. IEEE, 1989, pp. 59–62. — On page 1.

[38] Vincenzo Brancaccio. "A Systematic Literature Review Towards a Representative Feature-Model Benchmark". Master's Thesis. University of Ulm, 2022. DOI: 10.18725/OPARU-46759. — On page 67.

[39] Randal E. Bryant. "Binary Decision Diagrams". In: *Handbook of Model Checking*. Springer, 2018, pp. 191–217. DOI: 10.1007/978-3-319-10575-8_7. — On page 3.

[40] Jan Burchard, Tobias Schubert, and Bernd Becker. "Laissez-Faire Caching for Parallel #SAT Solving". In: *Proc. Int'l Conf. on Theory and Applications of Satisfiability Testing (SAT)*. Springer, 2015, pp. 46–61. — On pages 12, 26, 27, 33, 43, 49–51.

[41] Donald Chai and Andreas Kuehlmann. "A Fast Pseudo-Boolean Constraint Solver". In: *IEEE Trans. on Computer-Aided Design of Integrated Circuits and Systems* 24.3 (2005), pp. 305–317. — On pages 49, 52.

[42] Supratik Chakraborty, Daniel J Fremont, Kuldeep S Meel, Sanjit A Seshia, and Moshe Y Vardi. "On Parallel Scalable Uniform SAT Witness Generation". In: *Proc. Int'l Conf. on Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*. Springer, 2015, pp. 304–319. — On page 18.

[43] Supratik Chakraborty, Kuldeep S. Meel, and Moshe Y. Vardi. "A Scalable Approximate Model Counter". In: *Proc. Int'l Conf. on Principles and Practice of Constraint Programming (CP)*. Ed. by Christian Schulte. Springer, 2013, pp. 200–216. — On page 29.

[44] Sheng Chen and Martin Erwig. "Optimizing the Product Derivation Process". In: *Proc. Int'l Systems and Software Product Line Conf. (SPLC)*. IEEE, 2011, pp. 35–44. DOI: 10.1109/SPLC.2011.47. — On pages 1, 16, 18, 20, 23, 28, 57.

[45] Dmitry Chistikov, Rayna Dimitrova, and Rupak Majumdar. "Approximate Counting in SMT and Value Estimation for Probabilistic Programs". In: *Proc. Int'l Conf. on Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*. Berlin, Heidelberg: Springer, 2015, pp. 320–334. ISBN: 978-3-662-46681-0. — On page 51.

[46] Andreas Classen, Quentin Boucher, and Patrick Heymans. "A Text-Based Approach to Feature Modelling: Syntax and Semantics of TVL". In: *Science of Computer Programming (SCP)* 76.12 (2011). Special Issue on Software Evolution, Adaptability and Variability, pp. 1130–1143. — On page 65.

[47] Paul C Clements, John D McGregor, and Sholom G Cohen. *The Structured Intuitive Model for Product Line Economics (SIMPLE)*. Tech. rep. Carnegie Mellon University, 2005. — On pages 1–3, 8, 9, 15–18, 23, 33, 57.

[48] Anastasia Cmyrev and Ralf Reissing. "Efficient and Effective Testing of Automotive Software Product Lines". In: *Int'l J. Applied Science and Technology (IJAST)* 7.2 (2014). DOI: 10.14416/j.ijast.2014.05.001. — On page 1.

[49] Sholom Cohen. *Predicting When Product Line Investment Pays*. Tech. rep. Carnegie Mellon University, 2003. — On pages 18, 23.

[50] The Kernel Development Community. *KConfig Language*. Website: https://www.kernel.org/doc/html/latest/kbuild/kconfig-language.html. Accessed: 2024-01-30. 2018. — On page 27.

[51] Krzysztof Czarnecki and Chang Hwan Peter Kim. "Cardinality-Based Feature Modeling and Constraints: A Progress Report". In: *Proc. Int'l Workshop on Software Factories (SF)*. 2005, pp. 16–20. — On pages 18, 23, 49.

[52] Krzysztof Czarnecki and Andrzej Wąsowski. "Feature Diagrams and Logics: There and Back Again". In: *Proc. Int'l Systems and Software Product Line Conf. (SPLC)*. IEEE, 2007, pp. 23–34. — On pages 3, 8, 10, 21.

[53] Adnan Darwiche. "On the Tractable Counting of Theory Models and Its Application to Truth Maintenance and Belief Revision". In: *J. Applied Non-Classical Logics* 11.1-2 (2001), pp. 11–34. — On pages 41, 42.

[54] Adnan Darwiche. "A Compiler for Deterministic, Decomposable Negation Normal Form". In: *Proc. Conf. on Artificial Intelligence (AAAI)*. AAAI Press, 2002, pp. 627–634. — On pages 4, 12–14, 39, 41, 43, 45, 51, 55, 58.

[55] Adnan Darwiche. "SDD: A New Canonical Representation of Propositional Knowledge Bases". In: *International Joint Conferences on Artificial Intelligence*. AAAI Press, 2011, pp. 819–826. — On pages 12, 39.

[56] Adnan Darwiche and Pierre Marquis. "A Knowledge Compilation Map". In: *J. Artificial Intelligence Research (JAIR)* 17.1 (2002), pp. 229–264. — On pages 3, 4, 12, 13, 36–39, 58.

[57] Leonardo De Moura and Nikolaj Bjørner. "Z3: An Efficient SMT Solver". In: *Proc. Int'l Conf. on Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*. Springer, 2008, pp. 337–340. — On page 51.

[58] Leonardo De Moura and Nikolaj Bjørner. "Satisfiability Modulo Theories: An Appetizer". In: *Proc. Brazilian Symposium on Formal Methods (SBFM)*. Springer, 2009, pp. 23–36. — On page 65.

[59] Leonardo De Moura and Nikolaj Bjørner. "Satisfiability Modulo Theories: Introduction and Applications". In: *Comm. ACM* 54.9 (2011), pp. 69–77. — On page 51.

[60] Tom DeMarco. *Controlling Software Projects: Management, Measurement, and Estimates*. Prentice-Hall Inc., 1986. — On page 60.

[61] Stephanie Demers, Praveen Gopalakrishnan, and Latha Kant. "A Generic Solution to Software-in-the-Loop". In: *IEEE Military Communications Conference (MILCOM)*. 2007, pp. 1–6. DOI: 10.1109/MILCOM.2007.4455268. — On page 44.

[62] Deepak Dhungana, Paul Grünbacher, and Rick Rabiser. "The DOPLER Meta-Tool for Decision-Oriented Variability Modeling: A Multiple Case Study". In: *Automated Software Engineering* 18.1 (2011), pp. 77–114. — On pages 7, 50.

[63] William F. Dowling and Jean H. Gallier. "Linear-Time Algorithms For Testing the Satisfiability of Propositional Horn Formulae". In: *The Journal of Logic Programming* 1.3 (1984), pp. 267–284. ISSN: 0743-1066. DOI: https://doi.org/10.1016/0743-1066(84)90014-1. On page 38.

[64] Clemens Dubslaff, Nils Husung, and Nikolai Käfer. "Configuring BDD Compilation Techniques for Feature Models". In: *Proc. Int'l Systems and Software Product Line Conf. (SPLC)*. ACM, 2024, pp. 209–216. DOI: 10.1145/3646548.3676538. On pages 36, 37, 40, 59.

[65] Raphael Dunkel. "One Solver to Rule All Feature Models – Or Not? Addressing the Algorithm Selection Problem for #SAT". Bachelor's Thesis. University of Ulm, 2024. DOI: 10.18725/OPARU-52711. On pages 59, 67, 68.

[66] Raphael Dunkel. "Engineering and Selecting Machine Learning Features for Feature Models". To appear. Master's Thesis. University of Ulm, 2025. On page 59.

[67] Rafael Dutra, Kevin Laeufer, Jonathan Bachrach, and Koushik Sen. "Efficient Sampling of SAT Solutions for Testing". In: *Proc. Int'l Conf. on Software Engineering (ICSE)*. ACM, 2018, pp. 549–559. DOI: 10.1145/3180155.3180248. On page 10.

[68] Domenik Eichhorn, Tobias Pett, Nils Przigoda, Jessica Kindsvater, Christoph Seidl, and Ina Schaefer. "Coverage-Driven Test Automation for Highly-Configurable Railway Systems". In: *Proc. Int'l Working Conf. on Variability Modelling of Software-Intensive Systems (VaMoS)*. Odense, Denmark: ACM, 2023, pp. 23–30. ISBN: 9798400700019. DOI: 10.1145/3571788.3571794. On page 1.

[69] Emelie Engström and Per Runeson. "Software Product Line Testing - A Systematic Mapping Study". In: *J. Information and Software Technology (IST)* 53 (1 2011), pp. 2–13. DOI: 10.1016/j.infsof.2010.05.011. On page 1.

[70] Hafiyyan Sayyid Fadhlillah, Kevin Feichtinger, Philipp Bauer, Elene Kutsia, and Rick Rabiser. "V4rdiac: Tooling for Multidisciplinary Delta-Oriented Variability Management in Cyber-Physical Production Systems". In: *Proc. Int'l Workshop on Languages for Modelling Variability (MODEVAR)*. Proc. Int'l Systems and Software Product Line Conf. (SPLC). ACM, 2022, pp. 34–37. DOI: 10.1145/3503229.3547028. On pages 30, 50, 51, 65.

[71] Hélene Fargier and Pierre Marquis. "Extending the Knowledge Compilation Map: Krom, Horn, Affine and Beyond". In: *Proc. Conf. on Artificial Intelligence (AAAI)*. Vol. 2. AAAI Press. 2008, pp. 442–447. On page 37.

[72] Kevin Feichtinger, Johann Stöbich, Dario Romano, and Rick Rabiser. "TRAVART: An Approach for Transforming Variability Models". In: *Proc. Int'l Working Conf. on Variability Modelling of Software-Intensive Systems (VaMoS)*. ACM, 2021, 8:1–8:10. DOI: 10.1145/3442391.3442400. On pages 30, 50, 65.

[73] Kevin Feichtinger, Chico Sundermann, Thomas Thüm, and Rick Rabiser. "It's Your Loss: Classifying Information Loss During Variability Model Roundtrip Transformations". In: *Proc. Int'l Systems and Software Product Line Conf. (SPLC)*. ACM, 2022, pp. 67–78. DOI: 10.1145/3546932.3546990. On pages 50, 66.

[74] Alexander Felfernig, Gerhard Friedrich, Dietmar Jannach, Markus Stumptner, and Markus Zanker. "Configuration knowledge representations for Semantic Web applications". In: *Artif. Intell. Eng. Des. Anal. Manuf. (AIEDAM)* 17.1 (2003), pp. 31–50. DOI: 10.1017/S0890060403171041. On page 7.

[75] David Fernández-Amorós, Ruben Heradio Gil, and José Antonio Cerrada Somolinos. "Inferring Information From Feature Diagrams to Product Line Economic Models". In: *Proc. Int'l Systems and Software Product Line Conf. (SPLC)*. Springer, 2009, pp. 41–50. On pages 26, 57.

[76] David Fernández-Amorós, Ruben Heradio, José Antonio Cerrada, and Carlos Cerrada. "A Scalable Approach to Exact Model and Commonality Counting for Extended Feature Models". In: *IEEE Trans. on Software Engineering (TSE)* 40.9 (2014), pp. 895–910. On pages 1–3, 16–18, 22, 23, 26–28, 36, 40, 41, 50, 57.

[77] Johannes K. Fichte and Markus Hecher. *The Model Counting Competitions 2021-2023*. Tech. rep. Cornell University Library, 2025. DOI: 10.48550/arXiv.2504.13842. On pages 55, 59.

[78] Johannes K. Fichte, Markus Hecher, and Florim Hamiti. "The Model Counting Competition 2020". In: *ACM J. of Experimental Algorithmics (JEA)* 26 (2021). DOI: 10.1145/3459080. On pages 3, 11–13, 21, 26–28, 30, 43, 45, 49–51, 58, 59.

[79] Claudia Fritsch, Richard Abt, and Burkhardt Renz. "The Benefits of a Feature Model in Banking". In: *Proc. Int'l Systems and Software Product Line Conf. (SPLC)*. ACM, 2020. DOI: 10.1145/3382025.3414946. On page 7.

[80] Nils Froleyks, Marijn Heule, Markus Iser, Matti Järvisalo, and Martin Suda. "SAT Competition 2020". In: *Artificial Intelligence (AIJ)* 301 (2021), p. 103572. DOI: 10.1016/j.artint.2021.103572. On page 21.

[81] Nicolas Fußberger, Bo Zhang, and Martin Becker. "A Deep Dive into Android's Variability Realizations". In: *Proc. Int'l Systems and Software Product Line Conf. (SPLC)*. ACM, 2017, pp. 69–78. DOI: 10.1145/3106195.3106213. On page 7.

[82] José A Galindo, Mathieu Acher, Juan Manuel Tirado, Cristian Vidal, Benoit Baudry, and David Benavides. "Exploiting the Enumeration of All Feature Model Configurations: A New Perspective With Distributed Computing". In: *Proc. Int'l Systems and Software Product Line Conf. (SPLC)*. ACM, 2016, pp. 74–78. On page 57.

[83] José A. Galindo, David Benavides, Pablo Trinidad, Antonio-Manuel Gutiérrez-Fernández, and Antonio Ruiz-Cortés. "Automated Analysis of Feature Models: Quo Vadis?" In: *Computing* 101.5 (2019), pp. 387–433. DOI: 10.1007/s00607-018-0646-1. On page 25.

[84] José A. Galindo, José Miguel Horcas, Alexander Felfernig, David Fernández-Amorós, and David Benavides. "FLAMA: A Collaborative Effort to Build a New Framework for the Automated Analysis of Feature Models". In: *Proc. Int'l Systems and Software Product Line Conf. (SPLC)*. ACM, 2023, pp. 16–19. DOI: 10.1145/3579028.3609008. On pages 7, 10, 50, 65.

[85] Martin Gebser, Torsten Schaub, Sven Thiele, and Philippe Veber. "Detecting Inconsistencies in Large Biological Networks with Answer Set Programming". In: *Theory and Practice of Logic Programming (TPLP)* 11 (2011), pp. 323–360. DOI: 10.1017/S1471068410000554. On page 56.

[86] Guillaume Girol, Benjamin Farinier, and Sébastien Bardin. "Not All Bugs Are Created Equal, But Robust Reachability Can Tell the Difference". In: *Proc. Int'l Conf. on Computer Aided Verification (CAV)*. Berlin, Germany: Springer, 2021, pp. 669–693. ISBN: 978-3-030-81684-1. DOI: 10.1007/978-3-030-81685-8_32. — On page 11.

[87] Carla P. Gomes, Ashish Sabharwal, and Bart Selman. "Model Counting: A New Strategy for Obtaining Good Bounds". In: *Proc. Conf. on Artificial Intelligence (AAAI)*. AAAI Press, 2006, pp. 54–61. — On page 59.

[88] Oscar González-Rojas and Juan Tafurth. "Multi-cloud Services Configuration Based on Risk Optimization". In: *On the Move to Meaningful Internet Systems: OTM 2019 Conferences*. Ed. by Hervé Panetto, Christophe Debruyne, Martin Hepp, Dave Lewis, Claudio Agostino Ardagna, and Robert Meersman. Springer, 2019, pp. 733–749. — On page 7.

[89] Jilles van Gurp, Jan Bosch, and Mikael Svahnberg. "On the Notion of Variability in Software Product Lines". In: *Proc. Working Conf. on Software Architecture (WICSA)*. 2001, pp. 45–54. — On page 1.

[90] Albert Haag. ""Dealing" with configurable products in the SAP business suite". In: *Proc. Configuration Workshop (ConfWS)*. 2005, p. 68. — On page 50.

[91] Axel Halin, Alexandre Nuttinck, Mathieu Acher, Xavier Devroey, Gilles Perrouin, and Benoit Baudry. "Test Them All, Is It Worth It? Assessing Configuration Sampling on the JHipster Web Development Stack". In: *Empirical Software Engineering (EMSE)* 24.2 (2019), pp. 674–717. DOI: 10.1007/S10664-018-9635-4. — On page 8.

[92] Peter L Hammer and Eliezer Shlifer. "Applications of Pseudo-Boolean Methods to Economic Problems". In: *Theory and Decision* 1 (1971), pp. 296–308. — On page 56.

[93] Yvonne Heimowski. "Simplifying Feature Models for Better Scalability of #SAT Solvers". Bachelor's Thesis. University of Ulm, 2022. — On page 68.

[94] Marc Hentze, Tobias Pett, Chico Sundermann, Sebastian Krieter, Thomas Thüm, and Ina Schaefer. "Generic Solution-Space Sampling for Multi-Domain Product Lines". In: *Proc. Int'l Conf. on Generative Programming: Concepts & Experiences (GPCE)*. ACM, 2022. DOI: 10.1145/3564719.3568695. — On pages 3, 36, 44, 45, 60, 61, 64.

[95] Marc Hentze, Tobias Pett, Thomas Thüm, and Ina Schaefer. "Hyper Explanations for Feature-Model Defect Analysis". In: *Proc. Int'l Working Conf. on Variability Modelling of Software-Intensive Systems (VaMoS)*. ACM, 2021. DOI: 10.1145/3442391.3442406. — On page 64.

[96] Marc Hentze, Chico Sundermann, Thomas Thüm, and Ina Schaefer. "Quantifying the Variability Mismatch Between Problem and Solution Space". In: *Proc. Int'l Conf. on Model Driven Engineering Languages and Systems (MODELS)*. IEEE, 2022, pp. 322–333. DOI: 10.1145/3550355.3552411. — On pages 2, 3, 7, 9, 15, 16, 18, 21, 26, 27, 36, 41, 44, 45, 57, 60, 61, 64.

[97] Ruben Heradio, David Fernández-Amorós, José Antonio Cerrada, and Ismael Abad. "A Literature Review on Feature Diagram Product Counting and Its Usage in Software Product Line Economic Models". In: *Int'l J. Software Engineering and Knowledge Engineering (IJSEKE)* 23.08 (2013), pp. 1177–1204. — On pages 1–3, 8–10, 15–18, 23, 25, 26, 33, 36, 41, 57, 64.

[98] Ruben Heradio, David Fernández-Amorós, Christoph Mayr-Dorn, and Alexander Egyed. "Supporting the Statistical Analysis of Variability Models". In: *Proc. Int'l Conf. on Software Engineering (ICSE)*. IEEE, 2019, pp. 843–853. DOI: 10.1109/ICSE.2019.00091. — On pages 2, 3, 16, 17, 26, 57.

[99] Ruben Heradio, Héctor José Pérez-Morago, David Fernández-Amorós, Roberto Bean, Francisco Javier Cabrerizo, Carlos Cerrada, and Enrique Herrera-Viedma. "Binary Decision Diagram Algorithms to Perform Hard Analysis Operations on Variability Models". In: *Proc. Int'l Conf. on Intelligent Software Methodologies, Tools and Techniques (SOMET)*. IOS Press, 2016, pp. 139–154. — On pages 3, 50.

[100] Rubén Heradio-Gil, David Fernández-Amorós, José Antonio Cerrada, and Carlos Cerrada. "Supporting Commonality-Based Analysis of Software Product Lines". In: *IET Software* 5.6 (2011), pp. 496–509. — On pages 1, 18, 26, 28, 33, 36, 41, 57.

[101] Tobias Heß, Tobias Müller, Chico Sundermann, and Thomas Thüm. "ddueruem: A Wrapper for Feature-Model Analysis Tools". In: *Proc. Int'l Systems and Software Product Line Conf. (SPLC)*. ACM, 2022, pp. 54–57. DOI: 10.1145/3503229.3547032. — On pages 63, 64.

[102] Tobias Heß, Sean Niklas Semmler, Chico Sundermann, Jacobo Torán, and Thomas Thüm. "Towards Deterministic Compilation of Binary Decision Diagrams From Feature Models". In: *Proc. Int'l Systems and Software Product Line Conf. (SPLC)*. ACM, 2024, pp. 136–147. DOI: 10.1145/3646548.3672598. — On pages 40, 59, 63, 64.

[103] Tobias Heß, Chico Sundermann, and Thomas Thüm. "On the Scalability of Building Binary Decision Diagrams for Current Feature Models". In: *Proc. Int'l Systems and Software Product Line Conf. (SPLC)*. ACM, 2021, pp. 131–135. DOI: 10.1145/3461001.3474452. — On pages 17, 26, 36, 37, 41, 57, 63, 64.

[104] José M. Horcas, José A. Galindo, Lidia Fuentes, and David Benavides. "FM Fact Label". In: *Science of Computer Programming (SCP)* (2024), p. 103214. DOI: 10.1016/j.scico.2024.103214. — On page 65.

[105] José M. Horcas, José A. Galindo, Mónica Pinto, Lidia Fuentes, and David Benavides. "FM Fact Label: A Configurable and Interactive Visualization of Feature Model Characterizations". In: *Proc. Int'l Workshop on Languages for Modelling Variability (MODEVAR)*. Proc. Int'l Systems and Software Product Line Conf. (SPLC). ACM, 2022, pp. 42–45. DOI: 10.1145/3503229.3547025. — On page 50.

[106] José Miguel Horcas, Chico Sundermann, Malte Grave, José A. Galindo, Thomas Thüm, Rick Rabiser, and David Benavides. "Third Tutorial on the Universal Variability Language: Practical Extensions and Tool Ecosystem". In: *Proc. Int'l Systems and Software Product Line Conf. (SPLC)*. New York, NY, USA: ACM, 2025, p. 241. ISBN: 9798400720246. DOI: 10.1145/3744915.3749149. — On pages 66, 67.

[107] Arnaud Hubaux, Quentin Boucher, Herman Hartmann, Raphaël Michel, and Patrick Heymans. "Evaluating a Textual Feature Modelling Language: Four Industrial Case Studies". In: *Proc. Int'l Conf. on Software Language Engineering (SLE)*. Berlin, Heidelberg: Springer, 2011, pp. 337–356. ISBN: 978-3-642-19440-5. — On page 50.

[108] Arnaud Hubaux, Yingfei Xiong, and Krzysztof Czarnecki. "A User Survey of Configuration Challenges in Linux and eCos". In: *Proc. Int'l Workshop on Variability Modelling of Software-Intensive Systems (VaMoS)*. ACM, 2012, pp. 149–155. DOI: 10.1145/2110147.2110164. On page 1.

[109] Michael Huth and Mark Ryan. *Logic in Computer Science: Modelling and Reasoning About Systems*. Cambridge University Press, 2004. On pages 9, 11.

[110] Mikoláš Janota, Inês Lynce, and Joao Marques-Silva. "Algorithms for Computing Backbones of Propositional Formulae". In: *AI Communications* 28.2 (2015), pp. 161–177. On page 46.

[111] Jan Johannsen. "The Complexity of Pure Literal Elimination". In: *J. Autom. Reason.* 35 (2005), pp. 89–95. On page 46.

[112] Martin Fagereng Johansen, Øystein Haugen, and Franck Fleurey. "An Algorithm for Generating T-Wise Covering Arrays From Large Feature Models". In: *Proc. Int'l Systems and Software Product Line Conf. (SPLC)*. ACM, 2012, pp. 46–55. DOI: 10.1145/2362536.2362547. On page 9.

[113] Paulius Juodisius, Atrisha Sarkar, Raghava Rao Mukkamala, Michal Antkiewicz, Krzysztof Czarnecki, and Andrzej Wasowski. "Clafer: Lightweight Modeling of Structure, Behaviour, and Variability". In: *Computing Research Repository (CoRR)* (2018). On page 65.

[114] Kyo C. Kang, Sholom G. Cohen, James A. Hess, William E. Novak, and A. Spencer Peterson. *Feature-Oriented Domain Analysis (FODA) Feasibility Study*. Tech. rep. CMU/SEI-90-TR-21. Software Engineering Institute, 1990. On pages 3, 7.

[115] Kyo C. Kang, Jaejoon Lee, and Patrick Donohoe. "Feature-Oriented Product Line Engineering". In: *IEEE Software* 19.4 (2002), pp. 58–65. DOI: 10.1109/MS.2002.1020288. On page 7.

[116] Vladimir Klebanov, Norbert Manthey, and Christian Muise. "SAT-Based Analysis and Quantification of Information Flow in Programs". In: *Proc. Int'l Conf. on Quantitative Evaluation of Systems (QEST)*. Springer, 2013, pp. 177–192. On page 56.

[117] Will Klieber and Gihwon Kwon. "Efficient CNF Encoding for Selecting 1 From n Objects". In: *Proc. Int'l Workshop on Constraints in Formal Verification (CFV)*. Springer, 2007. On page 51.

[118] Alexander Knüppel, Thomas Thüm, Stephan Mennicke, Jens Meinicke, and Ina Schaefer. "Is There a Mismatch Between Real-World Feature Models and Product-Line Research?" In: *Proc. Europ. Software Engineering Conf./Foundations of Software Engineering (ESEC/FSE)*. ACM, 2017, pp. 291–302. DOI: 10.1145/3106237.3106252. On pages 8, 20, 24, 26–28, 30, 39, 43, 60.

[119] Tuukka Korhonen and Matti Järvisalo. *SharpSAT-TD in Model Counting Competitions 2021-2023*. Tech. rep. arXiv:2308.15819. Cornell University Library, 2023. DOI: 10.48550/arXiv.2308.15819. On pages 27, 55, 58, 59.

[120] Frédéric Koriche, Jean-Marie Lagniez, Pierre Marquis, and Samuel Thomas. "Knowledge Compilation for Model Counting: Affine Decision Trees". In: *Proc. Int'l Joint Conf. on Artificial Intelligence (IJCAI)*. AAAI Press, 2013. On pages 13, 38, 39.

[121] Matthias Kowal, Sofia Ananieva, and Thomas Thüm. "Explaining Anomalies in Feature Models". In: *Proc. Int'l Conf. on Generative Programming: Concepts & Experiences (GPCE)*. ACM, 2016, pp. 132–143. DOI: 10.1145/2993236.2993248. On pages 27, 43.

[122] Emily Kowalczyk, Myra B. Cohen, and Atif M. Memon. "Configurations in Android Testing: They Matter". In: *Proc. Int'l Workshop on Advances in Mobile App Analysis (A-Mobile)*. ACM, 2018, pp. 1–6. DOI: 10.1145/3243218.3243219. On page 1.

[123] Sebastian Krieter. "Efficient Interactive and Automated Product-Line Configuration". PhD thesis. University of Magdeburg, 2022. DOI: 10.25673/92625. On pages 25, 42.

[124] Sebastian Krieter, Rahel Arens, Michael Nieke, Chico Sundermann, Tobias Heß, Thomas Thüm, and Christoph Seidl. "Incremental Construction of Modal Implication Graphs for Evolving Feature Models". In: *Proc. Int'l Systems and Software Product Line Conf. (SPLC)*. ACM, 2021, pp. 64–74. DOI: 10.1145/3461001.3471148. On page 65.

[125] Sebastian Krieter, Kevin Feichtinger, José A. Galindo, David Benavides, Rick Rabiser, Chico Sundermann, and Thomas Thüm. "Second Tutorial on the Universal Variability Language". In: *Proc. Int'l Systems and Software Product Line Conf. (SPLC)*. ACM, 2023, p. 273. DOI: 10.1145/3579027.3609002. On pages 66, 67.

[126] Sebastian Krieter, Reimar Schröter, Thomas Thüm, Wolfram Fenske, and Gunter Saake. "Comparing Algorithms for Efficient Feature-Model Slicing". In: *Proc. Int'l Systems and Software Product Line Conf. (SPLC)*. ACM, 2016, pp. 60–64. DOI: 10.1145/2934466.2934477. On pages 3, 4, 45, 47, 60.

[127] Sebastian Krieter, Thomas Thüm, Sandro Schulze, Gunter Saake, and Thomas Leich. "YASA: Yet Another Sampling Algorithm". In: *Proc. Int'l Working Conf. on Variability Modelling of Software-Intensive Systems (VaMoS)*. ACM, 2020. DOI: 10.1145/3377024.3377042. On pages 9, 36.

[128] Sebastian Krieter, Thomas Thüm, Sandro Schulze, Reimar Schröter, and Gunter Saake. "Propagating Configuration Decisions With Modal Implication Graphs". In: *Proc. Int'l Conf. on Software Engineering (ICSE)*. ACM, 2018, pp. 898–909. DOI: 10.1145/3180155.3180159. On pages 8, 9, 36.

[129] Andreas Kübler, Christoph Zengler, and Wolfgang Küchlin. "Model Counting in Product Configuration". In: *Proc. Int'l Workshop on Logics for Component Configuration (LoCoCo)*. Open Publishing Association, 2010, pp. 44–53. DOI: 10.4204/EPTCS.29.5. On pages 1–4, 7–9, 15–23, 25–28, 36, 41, 50, 57, 60.

[130] Elias Kuiter, Tobias Heß, Chico Sundermann, Sebastian Krieter, Thomas Thüm, and Gunter Saake. "How Easy Is SAT-Based Analysis of a Feature Model?" In: *Proc. Int'l Working Conf. on Variability Modelling of Software-Intensive Systems (VaMoS)*. ACM, 2024, pp. 149–151. DOI: 10.1145/3634713.3634733. On pages 59, 65.

[131] Elias Kuiter, Sebastian Krieter, Chico Sundermann, Thomas Thüm, and Gunter Saake. "Tseitin or not Tseitin? The Impact of CNF Transformations on Feature-Model Analyses". In: *Proc. Int'l Conf. on Automated Software Engineering (ASE)*. ACM, 2022, 110:1–110:13. DOI: 10.1145/3551349.3556938. On pages 11, 26, 64, 65.

[132] Elias Kuiter, Chico Sundermann, Thomas Thüm, Tobias Heß, Sebastian Krieter, and Gunter Saake. "How Configurable is the Linux Kernel? Analyzing Two Decades of Feature-Model History". In: *Trans. on Software Engineering and Methodology (TOSEM)* (2025). DOI: 10.1145/3729423. — On pages 7, 23, 26, 31, 41, 51, 56, 58–60, 64.

[133] Jean-Marie Lagniez and Pierre Marquis. "An Improved Decision-DNNF Compiler". In: *Proc. Int'l Joint Conf. on Artificial Intelligence (IJCAI)*. International Joint Conferences on Artificial Intelligence, 2017, pp. 667–673. — On pages 4, 12–14, 26, 27, 39, 41, 43, 45, 47, 50, 51, 53, 55, 56.

[134] Yong Lai. "Approximate Model Counting by Partial Knowledge Compilation". In: *Computing Research Repository (CoRR)* (2018). — On pages 13, 50.

[135] Yong Lai, Kuldeep S Meel, and Roland HC Yap. "The Power of Literal Equivalence in Model Counting". In: *Proc. Conf. on Artificial Intelligence (AAAI)*. Vol. 35. 5. AAAI Press, 2021, pp. 3851–3859. — On page 46.

[136] Jia Hui Liang, Vijay Ganesh, Krzysztof Czarnecki, and Venkatesh Raman. "SAT-Based Analysis of Large Real-World Feature Models Is Easy". In: *Proc. Int'l Systems and Software Product Line Conf. (SPLC)*. Springer, 2015, pp. 91–100. — On pages 1–4, 9, 10, 21, 22, 26, 27, 41, 50, 57.

[137] Paolo Liberatore. "On the Complexity of Choosing the Branching Literal in DPLL". In: *J. Artificial Intelligence* 116.1-2 (2000), pp. 315–326. — On page 11.

[138] Lars Licha. "Cutting Edge T-Wise Sampling With ddnnife". Bachelor's Thesis. University of Ulm, 2023. — On pages 42, 56, 68.

[139] Stephen W. Liddle, David W. Embley, and Scott N. Woodfield. "Cardinality Constraints in Semantic Data Models". In: *Data and Knowledge Engineering (DKE)* 11.3 (1993), pp. 235–270. DOI: https://doi.org/10.1016/0169-023X(93)90024-J. — On page 56.

[140] Maria Isabel Limaylla-Lunarejo, Nelly Condori Fernández, and Miguel Rodríguez Luaces. "SimRE: A Requirements Similarity Tool for Software Product Lines". In: *Proc. ACM Symposium on Applied Computing (SAC)*. Catania International Airport, Catania, Italy: ACM, 2025, pp. 1473–1480. ISBN: 9798400706295. DOI: 10.1145/3672608.3707758. — On page 65.

[141] Jorn Lind-Nielsen. *BuDDy*. Website: http://buddy.sourceforge.net/manual/main.html. Accessed: 2020-03-02. 2020. — On page 39.

[142] Frank J. van der Linden, Klaus Schmid, and Eelco Rommes. *Software Product Lines in Action: The Best Industrial Practice in Product Line Engineering*. Springer, 2007. — On page 7.

[143] Jacob Loth. "Projected d-DNNF Compilation for Feature Models". Master's Thesis. University of Ulm, 2023. — On page 67.

[144] Jacob Loth, Chico Sundermann, Tobias Schrull, Thilo Brugger, Felix Rieg, and Thomas Thüm. "UVLS: A Language Server Protocol for UVL". In: *Proc. Int'l Systems and Software Product Line Conf. (SPLC)*. ACM, 2023, pp. 43–46. DOI: 10.1145/3579028.3609014. — On pages 30, 50, 51, 65, 66.

[145] Rafael Lotufo, Steven She, Thorsten Berger, Krzysztof Czarnecki, and Andrzej Wąsowski. "Evolution of the Linux Kernel Variability Model". In: *Proc. Int'l Systems and Software Product Line Conf. (SPLC)*. Springer, 2010, pp. 136–150. — On pages 7, 8, 20, 23, 24, 27, 31.

[146] Raúl Mazo, Cosmin Dumitrescu, Camille Salinesi, and Daniel Diaz. "Recommendation Heuristics for Improving Product Line Configuration Processes". In: *Recommendation Systems in Software Engineering*. Springer, 2014, pp. 511–537. On pages 1, 15–18, 20, 23, 57.

[147] Flávio Medeiros, Christian Kästner, Márcio Ribeiro, Rohit Gheyi, and Sven Apel. "A Comparison of 10 Sampling Algorithms for Configurable Systems". In: *Proc. Int'l Conf. on Software Engineering (ICSE)*. ACM, 2016, pp. 643–654. DOI: 10.1145/2884781.2884793. On pages 1, 16.

[148] Jens Meinicke, Thomas Thüm, Reimar Schröter, Fabian Benduhn, Thomas Leich, and Gunter Saake. *Mastering Software Variability With FeatureIDE*. Springer, 2017. DOI: 10.1007/978-3-319-61443-4. On pages 7, 8, 10, 26, 47, 53, 56, 57, 65.

[149] Marcílio Mendonça, Moises Branco, and Donald Cowan. "S.P.L.O.T.: Software Product Lines Online Tools". In: *Proc. Conf. on Object-Oriented Programming, Systems, Languages and Applications (OOPSLA)*. ACM, 2009, pp. 761–762. On pages 7, 27, 65.

[150] Marcílio Mendonça, Andrzej Wąsowski, and Krzysztof Czarnecki. "SAT-Based Analysis of Feature Models Is Easy". In: *Proc. Int'l Systems and Software Product Line Conf. (SPLC)*. Software Engineering Institute, 2009, pp. 231–240. On pages 21, 25.

[151] Raphaël Michel, Arnaud Hubaux, Vijay Ganesh, and Patrick Heymans. "An SMT-Based Approach to Automated Configuration". In: *Proc. Int'l Workshop on Satisfiability Modulo Theories (SMT)*. Vol. 20. EPiC Series in Computing. EasyChair, 2012, pp. 109–119. On page 51.

[152] Christian Muise, Sheila A. McIlraith, J. Christopher Beck, and Eric I. Hsu. "Dsharp: Fast d-DNNF Compilation with sharpSAT". In: *Advances in Artificial Intelligence*. Ed. by Leila Kosseim and Diana Inkpen. Springer, 2012, pp. 356–361. On pages 4, 13, 14, 27, 39, 41, 43, 45, 51, 55.

[153] Daniel-Jesus Munoz, Jeho Oh, Mónica Pinto, Lidia Fuentes, and Don Batory. "Uniform Random Sampling Product Configurations of Feature Models That Have Numerical Features". In: *Proc. Int'l Systems and Software Product Line Conf. (SPLC)*. ACM, 2019, pp. 289–301. DOI: 10.1145/3336294.3336297. On pages 18, 28.

[154] Daniel-Jesus Munoz, Mónica Pinto, Lidia Fuentes, and Don Batory. "Transforming Numerical Feature Models into Propositional Formulas and the Universal Variability Language". In: *J. Systems and Software (JSS)* 204 (2023). DOI: 10.1016/j.jss.2023.111770. On pages 49–51, 55, 60.

[155] Michael Nieke, Jacopo Mauro, Christoph Seidl, Thomas Thüm, Ingrid Chieh Yu, and Felix Franzke. "Anomaly Analyses for Feature-Model Evolution". In: *Proc. Int'l Conf. on Generative Programming: Concepts & Experiences (GPCE)*. ACM, 2018, pp. 188–201. DOI: 10.1145/3278122.3278123. On pages 28, 39.

[156] Michael Nieke, Gabriela Sampaio, Thomas Thüm, Christoph Seidl, Leopoldo Teixeira, and Ina Schaefer. "Guiding the Evolution of Product-Line Configurations". In: *Software and Systems Modeling (SoSyM)* 21 (1 2022), pp. 225–247. DOI: 10.1007/s10270-021-00906-w. On pages 26, 27, 31, 60.

[157] Masaaki Nishino, Norihito Yasuda, Shin-ichi Minato, and Masaaki Nagata. "Zero-Suppressed Sentential Decision Diagrams". In: *Proc. Conf. on Artificial Intelligence (AAAI)* 30 (2016). DOI: 10.1609/aaai.v30i1.10114. On page 39.

[158] Lina Ochoa, Oscar González-Rojas, and Thomas Thüm. "Using Decision Rules for Solving Conflicts in Extended Feature Models". In: *Proc. Int'l Conf. on Software Language Engineering (SLE)*. ACM, 2015, pp. 149–160. On page 51.

[159] Jeho Oh, Don Batory, Margaret Myers, and Norbert Siegmund. *Finding Product Line Configurations With High Performance by Random Sampling*. Tech. rep. University of Texas at Austin, Department of Computer Science, 2016. On page 28.

[160] Jeho Oh, Don Batory, Margaret Myers, and Norbert Siegmund. "Finding Near-Optimal Configurations in Product Lines by Random Sampling". In: *Proc. Int'l Symposium on Foundations of Software Engineering (FSE)*. 2017, pp. 61–71. DOI: 10.1145/3106237.3106273. On pages 9, 18.

[161] Jeho Oh, Paul Gazzillo, Don Batory, Marijn Heule, and Maggie Myers. *Uniform Sampling From Kconfig Feature Models*. Tech. rep. TR-19-02. University of Texas at Austin, Department of Computer Science, 2019. On pages 1–4, 9, 15, 16, 18, 22, 23, 25, 26, 39, 43, 57.

[162] Jeho Oh, Paul Gazzillo, Don Batory, Marijn Heule, and Margaret Myers. *Scalable Uniform Sampling for Real-World Software Product Lines*. Tech. rep. TR-20-01. University of Texas at Austin, Department of Computer Science, 2020. On page 9.

[163] Olesia Oliinyk, Kai Petersen, Manfred Schoelzke, Martin Becker, and Soeren Schneickert. "Structuring Automotive Product lines and Feature Models: An Exploratory Study at Opel". In: *Requirements Engineering* 22.1 (Mar. 2017), pp. 105–135. ISSN: 0947-3602. DOI: 10.1007/s00766-015-0237-z. On pages 7, 25, 26, 36, 41.

[164] Lukas Ostheimer. "Identification of Variance Driving Features in Feature Models". Bachelor's Thesis. University of Ulm, 2022. On page 68.

[165] Umut Oztok and Adnan Darwiche. "A Top-Down Compiler for Sentential Decision Diagrams". In: *Proc. Int'l Joint Conf. on Artificial Intelligence (IJCAI)*. AAAI Press, 2015, pp. 3141–3148. On pages 13, 37, 39, 45.

[166] Leonardo Passos, Marko Novakovic, Yingfei Xiong, Thorsten Berger, Krzysztof Czarnecki, and Andrzej Wąsowski. "A Study of Non-Boolean Constraints in Variability Models of an Embedded Operating System". In: *Proc. Int'l Workshop on Feature-Oriented Software Development (FOSD)*. Proc. Int'l Systems and Software Product Line Conf. (SPLC). ACM, 2011. DOI: 10.1145/2019136.2019139. On pages 50, 51, 60.

[167] Joaquin Peña, Michael G Hinchey, and Antonio Ruiz-Cortés. "Building the Core Architecture of a Multiagent System Product Line: With an example From a Future NASA Mission". In: *Proc. Int'l Workshop on Agent-Oriented Software Engineering (AOSE)*. Springer, 2006. On page 16.

[168] Héctor José Pérez Morago. "BDD Algorithms to Perform Hard Analysis Operations on Variability Models". PhD thesis. Universidad Nacional de Educación a Distancia, 2016. On pages 16–18, 28, 36, 37.

[169] Tobias Pett, Sebastian Krieter, Tobias Runge, Thomas Thüm, Malte Lochau, and Ina Schaefer. "Stability of Product-Line Sampling in Continuous Integration". In: *Proc. Int'l Working Conf. on Variability Modelling of Software-Intensive Systems (VaMoS)*. ACM, 2021. DOI: 10.1145/3442391.3442410. On page 28.

[170] Tobias Pett, Sebastian Krieter, Thomas Thüm, Malte Lochau, and Ina Schaefer. "AutoSMP: An Evaluation Platform for Sampling Algorithms". In: *Proc. Int'l Systems and Software Product Line Conf. (SPLC)*. ACM, 2021, pp. 41–44. DOI: 10.1145/3461002.3473073. On page 39.

[171] Tobias Pett, Thomas Thüm, Tobias Runge, Sebastian Krieter, Malte Lochau, and Ina Schaefer. "Product Sampling for Product Lines: The Scalability Challenge". In: *Proc. Int'l Systems and Software Product Line Conf. (SPLC)*. ACM, 2019, pp. 78–83. DOI: 10.1145/3336294.3336322. On pages 7, 26, 28, 31, 39, 43, 60.

[172] Quentin Plazar, Mathieu Acher, Gilles Perrouin, Xavier Devroey, and Maxime Cordy. "Uniform Sampling of SAT Solutions for Configurable Systems: Are We There Yet?" In: *Proc. Int'l Conf. on Software Testing, Verification and Validation (ICST)*. IEEE, 2019, pp. 240–251. DOI: 10.1109/ICST.2019.00032. On page 9.

[173] Klaus Pohl, Günter Böckle, and Frank J. van der Linden. *Software Product Line Engineering: Foundations, Principles and Techniques*. Springer, 2005. DOI: 10.1007/3-540-28901-1. On page 7.

[174] Richard Pohl, Kim Lauenroth, and Klaus Pohl. "A Performance Comparison of Contemporary Algorithmic Approaches for Automated Analysis Operations on Feature Models". In: *Proc. Int'l Conf. on Automated Software Engineering (ASE)*. IEEE, 2011, pp. 313–322. DOI: 10.1109/ASE.2011.6100068. On pages 3, 16, 17, 21, 26, 28, 57.

[175] Steven David Prestwich. "CNF Encodings". In: *Handbook of Satisfiability* 185 (2009), pp. 75–97. On pages 11, 30.

[176] Heiko Raab. "Exploiting d-DNNFs for Efficient Cardinality-Based Feature-Model Analyses". Bachelor's Thesis. University of Ulm, 2022. DOI: 10.18725/OPARU-43414. On page 67.

[177] Heiko Raab. "Incrementally Adapting d-DNNFs to Cope with Feature-Model Evolution". Master's Thesis. University of Ulm, 2023. DOI: 10.18725/OPARU-52288. On pages 42, 67.

[178] Rick Rabiser, Paul Grünbacher, and Deepak Dhungana. "Requirements for Product Derivation Support: Results from a Systematic Literature Review and an Expert Survey". In: *J. Information and Software Technology (IST)* 52.3 (2010), pp. 324–346. ISSN: 0950-5849. DOI: 10.1016/j.infsof.2009.11.001. On pages 25, 26.

[179] Jonathan Riesland. "Optimizing T-Wise Sampling on d-DNNFs with Objective Functions". To appear. Bachelor's Thesis. University of Ulm, 2024. On pages 42, 56, 68.

[180] Dario Romano, Kevin Feichtinger, Danilo Beuche, Uwe Ryssel, and Rick Rabiser. "Bridging the Gap Between Academia and Industry: Transforming the Universal Variability Language to Pure::Variants and Back". In: *Proc. Int'l Workshop on Languages for Modelling Variability (MODEVAR)*. Proc. Int'l Systems and Software Product Line Conf. (SPLC). ACM, 2022, pp. 123–131. DOI: 10.1145/3503229.3547056. On pages 30, 50, 51, 65.

[181] David Romero-Organvídez, José A Galindo, Chico Sundermann, José-Miguel Horcas, and David Benavides. "UVLHub: A Feature Model Data Repository Using UVL and Open Science Principles". In: *J. Systems and Software (JSS)* 215 (2024). DOI: 10.1016/j.jss.2024.112150. On pages 50, 51, 65, 66.

[182] Tian Sang, Fahiem Bacchus, Paul Beame, Henry A. Kautz, and Toniann Pitassi. "Combining Component Caching and Clause Learning for Effective Model Counting". In: *Proc. Int'l Conf. on Theory and Applications of Satisfiability Testing (SAT)*. Springer, 2004, pp. 20–28. On pages 3, 11, 12, 26, 27, 49, 51, 55, 58, 59.

[183] Tian Sang, Paul Beame, and Henry Kautz. "Heuristics for Fast Exact Model Counting". In: *Proc. Int'l Conf. on Theory and Applications of Satisfiability Testing (SAT)*. Springer, 2005, pp. 226–240. On pages 46, 59.

[184] Dominik Schiessl. "An Incremental #SAT Solver for Efficient Analysis of Feature Models". Master's Thesis. University of Ulm, 2022. On page 67.

[185] Sebastian Schlag, Tobias Heuer, Lars Gottesbüren, Yaroslav Akhremtsev, Christian Schulz, and Peter Sanders. "High-Quality Hypergraph Partitioning". In: *ACM J. of Experimental Algorithmics (JEA)* 27 (2023), pp. 1–39. On page 46.

[186] Clemens Schmid. "Parameterizations for Approximate #SAT Solvers". Bachelor's Thesis. University of Ulm, 2023. On pages 59, 67, 68.

[187] Pierre-Yves Schobbens, Patrick Heymans, and Jean-Christophe Trigaux. "Feature Diagrams: A Survey and a Formal Semantics". In: *Proc. Int'l Conf. on Requirements Engineering (RE)*. IEEE, 2006, pp. 136–145. DOI: 10.1109/RE.2006.23. On page 8.

[188] Reimar Schröter, Sebastian Krieter, Thomas Thüm, Fabian Benduhn, and Gunter Saake. "Feature-Model Interfaces: The Highway to Compositional Analyses of Highly-Configurable Systems". In: *Proc. Int'l Conf. on Software Engineering (ICSE)*. ACM, 2016, pp. 667–678. DOI: 10.1145/2884781.2884823. On page 8.

[189] Sandro Schulze, Thomas Thüm, Martin Kuhlemann, and Gunter Saake. "Variant-Preserving Refactoring in Feature-Oriented Software Product Lines". In: *Proc. Int'l Workshop on Variability Modelling of Software-Intensive Systems (VaMoS)*. ACM, 2012, pp. 73–81. DOI: 10.1145/2110147.2110156. On page 7.

[190] Sergio Segura. "Automated Analysis of Feature Models Using Atomic Sets". In: *Proc. Int'l Systems and Software Product Line Conf. (SPLC)*. Vol. 2. IEEE, 2008, pp. 201–207. On pages 25, 42.

[191] Sascha El-Sharkawy, Adam Krafczyk, and Klaus Schmid. "Analysing the KConfig Semantics and its Analysis Tools". In: *Proc. Int'l Conf. on Generative Programming: Concepts & Experiences (GPCE)*. ACM, 2015, pp. 45–54. DOI: 10.1145/2814204.2814222. On page 27.

[192] Sascha El-Sharkawy, Nozomi Yamagishi-Eichler, and Klaus Schmid. "Metrics for Analyzing Variability and its Implementation in Software Product Lines: A Systematic Literature Review". In: *J. Information and Software Technology (IST)* 106 (2019), pp. 1–30. DOI: 10.1016/j.infsof.2018.08.015. On page 1.

[193] Shubham Sharma, Rahul Gupta, Subhajit Roy, and Kuldeep S Meel. "Knowledge Compilation Meets Uniform Sampling". In: *Proc. Int'l Conf. on Logic for Programming, Artificial Intelligence, and Reasoning*. EasyChair, 2018, pp. 620–636. — On pages 4, 13, 16, 18, 36, 37, 42, 50.

[194] Shubham Sharma, Subhajit Roy, Mate Soos, and Kuldeep S Meel. "GANAK: A Scalable Probabilistic Exact Model Counter". In: *Proc. Int'l Joint Conf. on Artificial Intelligence (IJCAI)*. Vol. 19. AAAI Press, 2019, pp. 1169–1176. — On pages 12, 26, 27, 33, 43, 49, 50, 55, 58, 59.

[195] Arijit Shaw and Kuldeep S. Meel. *Model Counting in the Wild*. Tech. rep. Cornell University Library, 2024. eprint: 2408.07059 (cs.LO). URL: https://arxiv.org/abs/2408.07059. — On pages 11, 25.

[196] Carsten Sinz. "Towards an Optimal CNF Encoding of Boolean Cardinality Constraints". In: *Proc. Int'l Conf. on Principles and Practice of Constraint Programming (CP)*. Berlin, Heidelberg: Springer, 2005, pp. 827–831. ISBN: 978-3-540-32050-0. — On pages 55, 56, 61.

[197] Fabio Somenzi. *CUDD*. Website: https://github.com/vscosta/cudd. Accessed: 2020-06-13. 2020. — On page 39.

[198] Mate Soos and Kuldeep S. Meel. "Arjun: An Efficient Independent Support Computation Technique and its Applications to Counting and Sampling". In: *Proc. Int'l Conf. on Computer-Aided Design (ICCAD)*. ACM, 2022. DOI: 10.1145/3508352.3549406. — On page 45.

[199] David Speck, Markus Hecher, Daniel Gnad, Johannes K. Fichte, and Augusto B. Corrêa. "Counting and Reasoning with Plans". In: *Proc. Conf. on Artificial Intelligence (AAAI)* 39.25 (2025), pp. 26688–26696. DOI: 10.1609/aaai.v39i25.34871. — On pages 56, 61.

[200] Joshua Sprey and Chico Sundermann. "Computing Attribute Ranges for Partial Configurations With JavaSMT". Bachelor's Thesis. TU Braunschweig, 2018. DOI: 10.24355/dbbs.084-201807060926-0. — On page 68.

[201] Joshua Sprey, Chico Sundermann, Sebastian Krieter, Michael Nieke, Jacopo Mauro, Thomas Thüm, and Ina Schaefer. "SMT-Based Variability Analyses in FeatureIDE". In: *Proc. Int'l Working Conf. on Variability Modelling of Software-Intensive Systems (VaMoS)*. ACM, 2020. DOI: 10.1145/3377024.3377036. — On pages 7, 51, 65.

[202] Chico Sundermann. "Applications of #SAT Solvers on Product Lines". Master's Thesis. TU Braunschweig, 2020. DOI: 10.24355/dbbs.084-202009161329-0. — On page 69.

[B3] Chico Sundermann, Vincenzo Francesco Brancaccio, Elias Kuiter, Sebastian Krieter, Tobias Heß, and Thomas Thüm. "Collecting Feature Models from the Literature: A Comprehensive Dataset for Benchmarking". In: *Proc. Int'l Systems and Software Product Line Conf. (SPLC)*. ACM, 2024, pp. 54–65. DOI: 10.1145/3646548.3672590. — On pages vii, 1, 2, 4, 7, 8, 20, 23–26, 30–32, 53, 58–60, 65–67, 69, 190.

[203] Chico Sundermann, Kevin Feichtinger, Dominik Engelhardt, Rick Rabiser, and Thomas Thüm. "Yet Another Textual Variability Language? A Community Effort Towards a Unified Language". In: *Proc. Int'l Systems and Software Product Line Conf. (SPLC)*. ACM, 2021, pp. 136–147. DOI: 10.1145/3461001.3471145. — On pages 10, 50, 51, 65.

[204] Chico Sundermann, Kevin Feichtinger, José A. Galindo, David Benavides, Rick Rabiser, Sebastian Krieter, and Thomas Thüm. "Tutorial on the Universal Variability Language". In: *Proc. Int'l Systems and Software Product Line Conf. (SPLC)*. ACM, 2022, 260:1. DOI: 10.1145/3546932.3547024. — On page 66.

[205] Chico Sundermann, Tobias Heß, Dominik Engelhardt, Rahel Arens, Johannes Herschel, Kevin Jedelhauser, Benedikt Jutz, Sebastian Krieter, and Ina Schaefer. "Integration of UVL in FeatureIDE". In: *Proc. Int'l Workshop on Languages for Modelling Variability (MODEVAR)*. ACM, 2021, pp. 73–79. DOI: 10.1145/3461002.3473940. — On pages 30, 50, 51, 65, 66.

[B1] Chico Sundermann, Tobias Heß, Michael Nieke, Paul Maximilian Bittner, Jeffrey M. Young, Thomas Thüm, and Ina Schaefer. "Evaluating State-of-the-Art #SAT Solvers on Industrial Configuration Spaces". In: *Empirical Software Engineering (EMSE)* 28.2 (2023), p. 38. DOI: 10.1007/s10664-022-10265-9. — On pages vii, 2–4, 23, 25–29, 32, 34, 37, 41, 43, 45–47, 49, 50, 57–60, 63–65, 69, 141.

[206] Chico Sundermann, Tobias Heß, Michael Nieke, Paul Maximilian Bittner, Jeffrey M. Young, Thomas Thüm, and Ina Schaefer. "Evaluating State-of-the-Art #SAT Solvers on Industrial Configuration Spaces". In: *Proc. Software Engineering (SE)*. Gesellschaft für Informatik, 2024, pp. 67–68. DOI: 10.18420/sw2024_18. — On page 63.

[207] Chico Sundermann, Tobias Heß, Rahel Sundermann, Elias Kuiter, Sebastian Krieter, and Thomas Thüm. "Generating Feature Models with UVL's Full Expressiveness". In: *Proc. Int'l Workshop on Languages for Modelling Variability (MODEVAR)*. ACM, 2024, pp. 61–65. DOI: 10.1145/3646548.3676602. — On pages 50, 66.

[208] Chico Sundermann, Elias Kuiter, Tobias Heß, Heiko Raab, Sebastian Krieter, and Thomas Thüm. "On the Benefits of Knowledge Compilation for Feature-Model Analyses". In: *Proc. Int'l Systems and Software Product Line Conf. (SPLC)*. ACM, 2024, p. 217. DOI: 10.1145/3646548.3676540. — On page 63.

[A2C1] Chico Sundermann, Elias Kuiter, Tobias Heß, Heiko Raab, Sebastian Krieter, and Thomas Thüm. "On the Benefits of Knowledge Compilation for Feature-Model Analyses". In: *Annals of Mathematics and Artificial Intelligence (AMAI)* 92.5 (2024), pp. 1013–1050. DOI: 10.1007/s10472-023-09906-6. — On pages vii, 1–4, 8–10, 15, 17, 19, 21, 22, 24–27, 33–41, 43, 45–47, 50, 55, 57, 58, 60, 63–65, 69, 102.

[C3] Chico Sundermann, Jacob Loth, and Thomas Thüm. "Efficient Slicing of Feature Models via Projected d-DNNF Compilation". In: *Proc. Int'l Conf. on Automated Software Engineering (ASE)*. ACM, 2024, pp. 1332–1344. DOI: 10.1145/3691620.3695594. — On pages vii, 2–4, 22, 27, 33–35, 44–46, 48–50, 55, 56, 58–61, 64, 65, 67, 69, 236.

[A1] Chico Sundermann, Michael Nieke, Paul Maximilian Bittner, Tobias Heß, Thomas Thüm, and Ina Schaefer. "Applications of #SAT Solvers on Feature Models". In: *Proc. Int'l Working Conf. on Variability Modelling of Software-Intensive Systems (VaMoS)*. ACM, 2021. DOI: 10.1145/3442391.3442404. — On pages vii, 1–4, 8, 9, 11, 15–25, 33, 36, 40, 41, 44, 57, 58, 60, 69, 91.

[209] Chico Sundermann, Heiko Raab, Tobias Heß, Thomas Thüm, and Ina Schaefer. *Exploiting d-DNNFs for Repetitive Counting Queries on Feature Models*. Tech. rep. arXiv:2303.12383. Cornell University Library, 2023. DOI: 10.48550/arXiv.2303.12383. — On pages 41, 63.

[C2] Chico Sundermann, Heiko Raab, Tobias Heß, Thomas Thüm, and Ina Schaefer. "Reusing d-DNNFs for Efficient Feature-Model Counting". In: *Trans. on Software Engineering and Methodology (TOSEM)* 33.8 (2024). DOI: 10.1145/3680465. — On pages vii, 2–4, 27, 33–36, 40, 42–44, 46, 50, 55, 56, 58–61, 63–65, 67, 69, 203.

[B2] Chico Sundermann, Thomas Thüm, and Ina Schaefer. "Evaluating #SAT Solvers on Industrial Feature Models". In: *Proc. Int'l Working Conf. on Variability Modelling of Software-Intensive Systems (VaMoS)*. ACM, 2020. DOI: 10.1145/3377024.3377025. — On pages vii, 2, 4, 25–30, 32, 34, 37, 41, 50, 57–60, 63, 180.

[C4] Chico Sundermann, Stefan Vill, Elias Kuiter, Sebastian Krieter, Thüm, and Matthias Tichy. *Pseudo-Boolean d-DNNF Compilation for Expressive Feature Modeling Constructs*. Tech. rep. arXiv:2505.05976. Submitted to TSE. Cornell University Library, 2025. DOI: 10.48550/arXiv.2505.05976. — On pages vii, 2–4, 27, 33–35, 49–56, 58, 60, 65–67, 69, 250.

[210] Chico Sundermann, Stefan Vill, Thomas Thüm, Kevin Feichtinger, Prankur Agarwal, Rick Rabiser, José A. Galindo, and David Benavides. "UVLParser: Extending UVL With Language Levels and Conversion Strategies". In: *Proc. Int'l Systems and Software Product Line Conf. (SPLC)*. ACM, 2023, pp. 39–42. DOI: 10.1145/3579028.3609013. — On pages 21, 50, 51, 53, 65, 66.

[211] Reinhard Tartler, Julio Sincero, Wolfgang Schröder-Preikschat, and Daniel Lohmann. "Dead or Alive: Finding Zombie Features in the Linux Kernel". In: *Proc. Int'l Workshop on Feature-Oriented Software Development (FOSD)*. ACM, 2009, pp. 81–86. — On page 1.

[212] Samuel Teuber and Alexander Weigl. "Quantifying Software Reliability via Model-Counting". In: *Proc. Int'l Conf. on Quantitative Evaluation of Systems (QEST)*. Ed. by Alessandro Abate and Andrea Marin. Cham, Switzerland: Springer, 2021, pp. 59–79. ISBN: 978-3-030-85172-9. — On pages 11, 56, 61.

[213] Thomas Thüm. "A BDD for Linux? The Knowledge Compilation Challenge for Variability". In: *Proc. Int'l Systems and Software Product Line Conf. (SPLC)*. ACM, 2020. DOI: 10.1145/3382025.3414943. — On page 13.

[214] Thomas Thüm, Sven Apel, Christian Kästner, Ina Schaefer, and Gunter Saake. "A Classification and Survey of Analysis Strategies for Software Product Lines". In: *ACM Computing Surveys (CSUR)* 47.1 (2014), 6:1–6:45. DOI: 10.1145/2580950. — On page 16.

[215] Marc Thurley. "sharpSAT - Counting Models With Advanced Component Caching and Implicit BCP". In: *Proc. Int'l Conf. on Theory and Applications of Satisfiability Testing (SAT)*. Springer, 2006, pp. 424–429. — On pages 3, 11, 12, 26, 27, 33, 43, 45, 49–52, 55, 56, 59.

[216] Takahisa Toda and Takehide Soh. "Implementing Efficient All Solutions SAT Solvers". In: *ACM J. of Experimental Algorithmics (JEA)* 21.1 (2016), 1.12:1–1.12:44. DOI: 10.1145/2975585. — On page 39.

[217] Pablo Trinidad, David Benavides, and Antonio Ruiz Cortés. "Isolated Features Detection in Feature Models". In: *Proc. Int'l Conf. on Advanced Information Systems Engineering (CAiSE)*. Springer, 2006. — On page 18.

[218] Leslie G Valiant. "The Complexity of Computing the Permanent". In: *Theoretical Computer Science* 8.2 (1979), pp. 189–201. — On pages 2, 3, 11, 22, 23, 25.

[219] Guy Van den Broeck and Adnan Darwiche. "On the Role of Canonicity in Knowledge Compilation". In: *Proc. Conf. on Artificial Intelligence (AAAI)* 29.1 (2015). DOI: 10.1609/aaai.v29i1.9423. On page 38.

[220] Ángel Jesús Varela-Vaca, Rafael M. Gasca, Jose Antonio Carmona-Fombella, and María Teresa Gómez-López. "AMADEUS: Towards the AutoMAteD SecUrity TeSting". In: *Proc. Int'l Systems and Software Product Line Conf. (SPLC)*. ACM, 2020. DOI: 10.1145/3382025.3414952. On page 1.

[221] Stefan Vill. "Language Levels for the Universal Variability Language: An Extension Mechanism and Conversion Strategies". Bachelor's Thesis. University of Ulm, 2022. DOI: 10.18725/OPARU-47708. On page 68.

[222] Stefan Vill. "Pseudo Boolean d-DNNF Compilation for Analyzing Expressive Feature-Model Formats". Master's Thesis. University of Ulm, 2025. DOI: 10.18725/OPARU-56664. On page 67.

[223] Jinyan Wang, Wenxiang Gu, Minghao Yin, and Dongxiu Wang. "MCN and MO: Two Heuristic Strategies in Knowledge Compilation Using Extension Rule". In: *International Conference on Signal Processing Systems*. 2009, pp. 389–393. DOI: 10.1109/ICSPS.2009.99. On page 38.

[224] Shuai Wang, Shaukat Ali, Tao Yue, and Marius Liaaen. "Using Feature Model to Support Model-Based Testing of Product Lines: An Industrial Case Study". In: *Proc. Int'l Conf. on Quality Software (QSIC)*. 2013, pp. 75–84. DOI: 10.1109/QSIC.2013.51. On pages 41, 50.

[225] Wei Wei and Bart Selman. "A New Approach to Model Counting". In: *Proc. Int'l Conf. on Theory and Applications of Satisfiability Testing (SAT)*. Springer, 2005, pp. 324–339. On page 29.

[226] Len Wozniak and Paul Clements. "How Automotive Engineering Is Taking Product Line Engineering to the Extreme". In: *Proc. Int'l Systems and Software Product Line Conf. (SPLC)*. SPLC '15. ACM, 2015, pp. 327–336. DOI: 10.1145/2791060.2791071. On page 7.

[227] Suwei Yang and Kuldeep S. Meel. "Engineering an Exact Pseudo-Boolean Model Counter". In: *Proc. Conf. on Artificial Intelligence (AAAI)* 38.8 (2024), pp. 8200–8208. DOI: 10.1609/aaai.v38i8.28660. On page 51.

[228] Olivier Zeyen, Maxime Cordy, Gilles Perrouin, and Mathieu Acher. "Preprocessing Is What You Need: Understanding and Predicting the Complexity of SAT-based Uniform Random Sampling". In: *Proc. Int'l Conf. on Formal Methods in Software Engineering (FormaliSE)*. ACM, 2024, pp. 23–32. DOI: 10.1145/3644033.3644371. On page 59.

Original Core Publications

In the following, we provide a list of the original publications that assemble the main contributions of this thesis.

C.1 VaMoS'21: Applications of #SAT Solvers on Feature Models

[A1] Chico Sundermann, Michael Nieke, Paul Maximilian Bittner, Tobias Heß, Thomas Thüm, and Ina Schaefer. "Applications of #SAT Solvers on Feature Models". In: *Proc. Int'l Working Conf. on Variability Modelling of Software-Intensive Systems (VaMoS)*. ACM, 2021. DOI: 10.1145/3442391.3442404

© 2021 Copyright is held by authors and they preserve rights to include a partial or complete version of the paper in a dissertation. Check https://doi.org/10.1145/3442391.3442404 for the published version.

Applications of #SAT Solvers on Feature Models

Chico Sundermann
University of Ulm, Germany

Michael Nieke
TU Braunschweig, Germany

Paul Maximilian Bittner
University of Ulm, Germany

Tobias Heß
University of Ulm, Germany

Thomas Thüm
University of Ulm, Germany

Ina Schaefer
TU Braunschweig, Germany

ABSTRACT

Product lines are ubiquitous for managing variable systems. The variability of a product line is typically described in terms of a feature model. Analyzing a feature model gives insight into various aspects, such as the validity of a configuration of features. Several analyses have been considered that require computing the number of valid configurations which proves highly inefficient when using regular SAT solvers. A #SAT solver computes the number of satisfying variable assignments of a propositional formula and is specifically optimized for the aforementioned analyses. In this work, we summarize and unify the state-of-the-art on #SAT-based feature-model analyses and propose a variety of new #SAT-based analyses. We provide an exhaustive catalogue for applications of #SAT for feature models serving as a reference for researchers and practitioners. Furthermore, we show that all these 21 applications are based on only three different #SAT queries. Thus, future research can focus on providing solutions and optimizations for those three queries to accelerate #SAT-based applications.

CCS CONCEPTS

• **General and reference** → *Surveys and overviews*; • **Software and its engineering → Software configuration management and version control systems**; *Constraint and logic languages*; *Automated static analysis*; *Software evolution*; • **Hardware** → *Theorem proving and SAT solving*.

KEYWORDS

Feature Models, Product Lines, Model Counting, Configuration Counting, #SAT, #SAT Applications, Feature-Model Analysis

ACM Reference Format:
Chico Sundermann, Michael Nieke, Paul Maximilian Bittner, Tobias Heß, Thomas Thüm, and Ina Schaefer. 2021. Applications of #SAT Solvers on Feature Models. In *15th International Working Conference on Variability Modelling of Software-Intensive Systems (VaMoS'21), February 9–11, 2021, Krems, Austria.* ACM, New York, NY, USA, 10 pages. https://doi.org/10.1145/3442391.3442404

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
VaMoS'21, February 9–11, 2021, Krems, Austria
© 2021 Copyright held by the owner/author(s). Publication rights licensed to ACM.
ACM ISBN 978-1-4503-8824-5/21/02…$15.00
https://doi.org/10.1145/3442391.3442404

1 INTRODUCTION

Product lines are widely used to manage families of similar products [10, 13, 32]. Each product is composed from a set of distinct units, called *features*, that are reused across products [56]. A product can be identified by the set of features it implements, referred to as its *configuration*. The most common way to specify the set of valid configurations are *feature models* [8, 10, 20].

Managing a product line is challenging due to the high number of constraints imposed by the feature model [10]. Various analyses have been proposed, such as checking the validity of a configuration [8, 10, 20, 25, 43, 51–54, 57]. Industrial feature models, such as the Linux kernel, contain thousands of features and constraints [38, 59]. Furthermore, these systems often grow significantly over time [26, 33, 38, 40, 59]. To properly manage a product line over time, it is necessary to control its variability. Tom DeMarco amongst others stated "*you can't control what you can't measure*" [22]. We propose using the number of valid configurations as a metric to measure the variability of a product line. Several analyses dependent on the number of valid configurations have been proposed, such as uniform random sampling [44, 47, 48, 55]. Typically, a feature model is translated to propositional logic to analyze it [8, 10, 61]. While it is possible to count assignments using satisfiability-based tools, such as SAT solvers, this typically comes with major scalability issues [48, 52, 59, 63].

Counting satisfying assignments of a propositional formula is known as #SAT and well researched [14, 21, 35, 39, 62]. Yet, possible applications of #SAT technology for feature models have been researched sparsely [16, 23, 24, 28, 32, 38]. Heradio et al. [28, 32] propose several applications based on computing the number of valid configurations limited to estimations of the economic benefits of a product line. Fernandez et al. [24] provide a smaller scale survey with only five applications which are limited to computing the number of valid configurations that contain a certain feature. Other research that proposes using the number of valid configurations just considers very few applications [10, 16, 23, 30, 38, 41, 52, 59]. Gathering an overview for applications based on the number of valid configurations currently requires a large time commitment for researchers and practitioners. Furthermore, proposed applications are often vaguely defined, lack formal definitions, or miss out on feasible algorithms to compute them. This induces additional effort to (1) generally interpret the application, (2) resolve ambiguities, and (3) compute actual results. We consider each of those aspects in our work to make applications of #SAT solvers on feature models more accessible with the following three major contributions.

First, we survey applications of #SAT on feature models considered in the literature. The goal is to gather existing applications with detailed and consistent descriptions in order to significantly

reduce the effort for research. Overall, we identified twelve distinct applications considered in the literature.

Second, we propose nine new #SAT-based applications to show additional possibilities and further reveal the currently unused potential of applying #SAT for feature-model analysis. The newly considered applications are mainly inspired by the cooperation with industry partners.

Third, we identified three different queries to a #SAT solver which provide the basis for every single of the 21 considered applications. In particular, these queries consist of computing (1) the number of valid configurations for the entire feature model (cf. Section 4), (2) the number of valid configurations including a particular feature (cf. Section 5), and (3) the number of remaining valid configurations of a partial configuration (cf. Section 6). For each of the three queries, we provide a solution that can easily be applied to off-the-shelf #SAT solvers. As these three queries form the basis for all identified applications, optimizations can focus on these three queries alone.

In summary, we provide an expert survey for applications dependent on computing the number of valid configurations of a feature model. Our work motivates further research and applying #SAT technology to feature models and makes required information vastly more accessible as there has not been a large-scale survey of #SAT-based applications before.

2 RUNNING EXAMPLE

In this section, we introduce a feature model that we use in the following sections to explain and motivate applications for #SAT. Fig. 1 shows the feature model of a simplified car. Each car requires a `Carbody` (denoted by MANDATORY) and exactly one `Gearbox`-type (denoted by ALTERNATIVE). Cars may contain a `Radio` that can be further configured (denoted by OPTIONAL). Each `Radio` may contain a `USB`-port, `CD`-port, or both (denoted by the OR group of `Radio's` OPTIONAL child `Ports`). The cross-tree constraint `Navigation` $\Rightarrow$ `USB` ensures that each car with `Navigation` has an `USB` port.

Both, the tree hierarchy and the cross-tree constraints limit the set of valid configurations. Computing the cardinality of the set of valid configurations is trivial for feature models without cross-tree constraints [32]. For such models, each sub-tree can be analyzed separately as there are no interdependencies between features from different sub-trees. Thus, the number of valid configurations can be computed with a linear-time traversal of the feature model by applying certain rules for each relationship type (e.g., the number of valid configuration of an alternative group is the sum of its features) [32]. However, for feature models with cross-tree constraints simply traversing the tree typically leads to faulty results as the restrictions imposed by the cross-tree constrains are disregarded.

3 BACKGROUND

SAT describes the problem of computing the satisfiability of a propositional formula. A propositional formula F is defined over a set of variables $vars(F)$ and consists of literals and logical operators $\neg$ (not), $\vee$ (or), and $\wedge$ (and). Each literal corresponds to exactly one variable $v \in vars(F)$. An assignment $\alpha : vars(F) \rightarrow \{\top, \bot\}$ maps variables $v \in vars(F)$ to the truth values $\top$ (true) and $\bot$ (false). We denote the number of variables which α assigns to truth values by $|\alpha|$. If $|\alpha| = |vars(F)|$, we call α *total*. Otherwise, we call it *partial*.

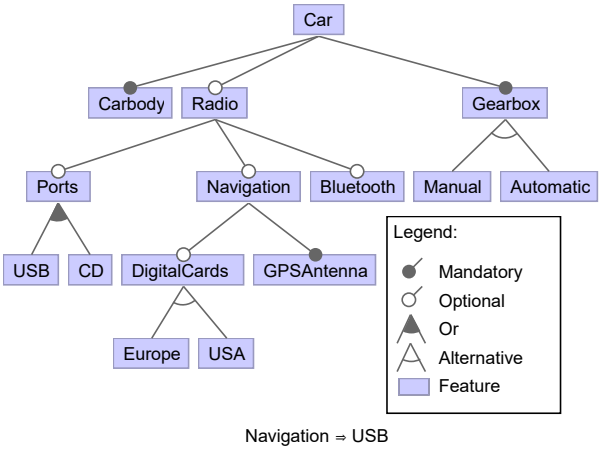


Figure 1: Simplified Car Feature Model (adapted from [5])

We lift assignments α to formulas F such that $\alpha(F)$ evaluates the formula F under α. If $\alpha(F) = \top$, we call α a *satisfying* assignment for F. Formally, the SAT problem corresponds to deciding whether an α exists with $\alpha(F) = \top$ for a given formula F.

#SAT, instead, is the problem of computing the number of satisfying assignments for a propositional formula F. Let A be the set of all total assignments over $vars(F)$. As each variable can be assigned either to $\top$ or $\bot$, there are $|A| = 2^{|vars(F)|}$ total assignments. Typically, not every assignment α satisfies the formula F. A #SAT solver computes the number of assignments in A that satisfy F (i.e., $\#F = \left|\{\alpha \in A \mid \alpha(F) = \top\}\right|$). #SAT is widely believed to be a computationally harder problem than SAT [14, 65]. It is obvious that #SAT is at least as hard as SAT considering it is trivial to determine whether a formula is satisfiable after computing the number of satisfying assignments [34].

We define a feature model as a tuple $FM = (FEATS, CONST)$ over a set of features $FEATS$ and a set of boolean constraints $CONST$ which includes tree and cross-tree constraints. A configuration $C = (FM, I, E)$ consists of a set of included features I and excluded features E. A feature cannot be included and excluded in the same configuration (i.e., $I \cap E = \emptyset$). If each feature is either included or excluded in C (i.e., $I \cup E = FEATS$), we call C *total* [10]. Otherwise, C is *partial*. We consider a configuration $C = (FM, I, E)$ to be *valid* if the sets of included I and excluded E features violate no constraints in $CONST$. Checking whether a configuration violates a subset of the constraints is typically infeasible by hand and, thus, requires efficient tool support [16, 31, 57].

Instead of developing new ad-hoc algorithms to analyze feature models, they are commonly translated to equivalent propositional formulas [8, 20, 25, 43, 51–54]. A formula F_{FM} is *equivalent* to a feature model FM ($F_{FM} \equiv FM$) if the following two properties hold: (1) for each configuration $C = (FM, \{i_1, ...i_n\}, \{e_1, ..., e_m\})$ and an assignment α, with $\alpha(i_{k=1,...,n}) = \top$ and $\alpha(e_{j=1,...,m}) = \bot$, $\alpha(F_{FM}) = \top$ iff C is valid and (2) $vars(F_{FM}) = FEATS$. For each feature model an equivalent propositional formula exists [10, 43]. The number of satisfying assignments of an equivalent formula F_{FM} is equal to the number of valid configurations by definition (i.e., $\#F_{FM} = \#FM$). Thus, invoking a #SAT solver with F_{FM} as input can be used to compute the number of valid configurations $\#FM$. In

Application	Source
Void feature model	Benavides et al. [10]
Variability factor	Benavides et al. [9], Fernandez et al. [24], Heradio et al. [28, 32]
Variability of feature subsets	**New**
Maintainability prediction	Bagheri et al. [7]
Variability reduction	**New**
Rating errors	Kübler et al. [38]
Degree of orthogonality	Czarnecki et al. [19] Benavides et al. [10]
Configuration relevance	**New**
Cost savings of a product line	Clements et al. [17], Heradio et al. [28]

Table 1: Applications - Cardinality of Feature Models

the following, we present three queries which we use as input for #SAT solvers to acquire results for all 21 considered applications.

4 CARDINALITY OF FEATURE MODELS

The most basic query to analyze feature models using #SAT solvers is to compute the number of valid configurations of the entire feature model. We refer to the cardinality of the set of valid configurations as the *cardinality of the feature model.*

Definition 1 (Cardinality of a Feature Model). *Let FM be a feature model and VC the set of valid configurations for FM. The cardinality of FM is defined by the cardinality of VC:*

$$\#FM = |VC|$$

For a feature model FM translated to an equivalent propositional formula F_{FM} (i.e., $FM \equiv F_{FM}$), $\#FM = \#F_{FM}$ holds by definition. Thus, $\#FM$ can be computed by invoking a #SAT solver on F_{FM} without further adaptations. As the translation of feature models to propositional formulas is well-researched [8, 10, 43, 61] and we propose using off-the-shelf #SAT solvers, this is trivially applicable. Table 1 provides an overview of applications based on the cardinality of feature models.

Void Feature Model. A feature model is considered void if it induces no valid configurations [10]. Computing its cardinality implicitly detects whether the feature model is void.

$$\#FM = 0 \Leftrightarrow FM \text{ is void} \tag{1}$$

In general, computing the voidness of a feature model with a SAT solver should be more efficient. However, if a feature model's cardinality is required anyways, an additional SAT call can be saved.

Variability Factor. The variability factor of a feature model describes the relative share of configurations that are valid compared to all configurations [9, 24, 28, 32]. Disregarding all restrictions, every combination of features induces a valid configuration. In this case, there are $2^{|FEATS|}$ configurations. The variability factor of a feature model FM is computed as follows:

$$VariabilityFactor(FM) = \frac{\#FM}{2^{|FEATS|}} \tag{2}$$

The variability factor lies between zero and one. A variability factor close to zero indicates a highly restricted feature model, while a feature model with no restrictions has a variability factor of one [28, 32]. Benavides et al. [9] argue that the variability factor can be used to estimate the benefits of a product-line approach as a small variability factor may indicate that developing standalone products is more beneficial. An unexpected variability factor may indicate design errors.

Our running example contains 15 features, only 1 cross-tree constraint, and induces 42 valid configurations. The variability factor $\frac{42}{2^{15}} = \frac{42}{32768} = 0.0013$ indicates that only a small fraction of all configurations is valid. This shows that the feature-model hierarchy limits the set of valid configurations by a large margin.

The strong limitations of the tree structure may make it difficult to grasp the impact of cross-tree constraints. We propose to simplify grasping the impact by introducing the *cross-tree variability factor* which compares the cardinality of a feature model with cross-tree constraints (FM) and without ($FM_{\overline{CTC}}$).

$$VariabilityFactor_{CTC}(FM) = \frac{\#FM}{\#FM_{\overline{CTC}}} \tag{3}$$

A larger cross-tree variability factor indicates that approximating the cardinality of the feature model by disregarding cross-tree constraints provides more accurate results. Without cross-tree constraints, our running example induces 66 valid configurations. Thus, the variability factor of cross-tree constraints is $\frac{42}{66} = 0.636$ which shows that the cross-tree constraints do not limit the number of valid configurations by a large margin.

Variability of Feature Subsets. Analyzing large feature models is computationally expensive. In some instances, it may be sufficient to focus analyses on specific parts of the feature model. We propose using the cardinality of a feature model to compute the variability of only a feature subset (e.g., a feature model sub-tree). For instance, this can be used to compute the cardinality of a feature model disregarding *abstract* features. Abstract features are irrelevant for the actual products and are used to structure the feature model [61]. The variability of a feature subset can also be used to find parts with a particularly high or low variability. Other analyses can then focus on identified parts. Furthermore, an unexpected variability degree for a subset may indicate modeling errors.

Given a set of features $S \subseteq FEATS$, the idea is to compute the number of different combinations of S that appear in the set of valid configurations. Consider the following example: Let $S = \{A, C\}$ be a feature subset and $I_1 = \{A, B, D\}$, $I_2 = \{A, C, D\}$, $I_3 = \{A, B, C\}$, and $I_4 = \{A, B, C, D\}$ the valid configurations induced by the feature model. While there are overall four valid configurations, only two different combinations of the features A, C appear ($\{A\}$ and $\{A, C\}$). In this case, the variability of the feature subset S is two. To compute the variability of a feature subset, we propose to use a combination of feature-model slicing and #SAT. Slicing can be used to remove features while preserving the restrictions that were imposed by the removed feature [1, 36]. After slicing out all features that are not part of the subset (i.e., $FEATS \setminus S$), we obtain a feature model that only contains features that appear in S but remains all original restrictions. Computing the cardinality of the resulting feature model is equal to the variability of S in the feature model FM.

$$SubsetVariability(FM, S) = \#(Slice(FM, FEATS \setminus S)) \tag{4}$$

In our example, the sub-tree induced by the feature `Radio` may be our subset S. The sub-tree induces 21 different variants. With this number, we can see that the main variability results from S, as the rest of the feature model only induces $\frac{42}{21} = 2$ variants which result from the alternative between `Automatic` and `Manual`.

Maintainability Prediction. Bagheri et al. [6] predict the maintainability of a product line using different metrics about the feature model such as the number of features or its cardinality. Such predictions using structural metrics allows assessing the maintainability of a product line in early stages of development. The insights can be used to preserve maintainability in later stages by effectively reducing the complexity early. Bagheri et al. examine statistical correlations between ten metrics and the maintainability. According to their evaluation, a high cardinality of the feature model strongly indicates a high complexity of the product line. With the gathered insights, they propose a prediction model to estimate the maintainability. The model includes the cardinality of a feature model as one of four metrics.

Variability Reduction. We propose to use the cardinality of a feature model to examine the impact of a change to the feature model. We call the change in cardinality from FM to an adaptation FM' variability reduction.

$$VariabilityReduction(FM, FM') = \#FM - \#FM' \tag{5}$$

One use-case of this metric is to identify undesired changes to the product line. If the cardinality of a feature model changes in an unexpected way, this is an indicator for a faulty edit. For example, an additional constraint that does not change the cardinality of the feature model is redundant [38]. Also, new constraints may unexpectedly reduce the cardinality by a large margin. In this case, the edits can be re-evaluated and design flaws can be prevented. For instance, for our running example, a company sells cars with `Automatic` exclusively to the USA. Thus, developers want to introduce a new cross-tree constraint $\texttt{Automatic} \Rightarrow \texttt{USA}$. The resulting feature model FM' induces 25 valid configurations. The original 42 valid configurations can be separated in 21 configurations with `Automatic` and 21 configurations with `Manual`. As the introduced constraint does not remove any valid configuration with `Manual`, there are only 4 valid configurations with `Automatic` left. This information can be used by developers to reconsider the changes. A possible alternative may be $\texttt{DigitalCards} \wedge \texttt{Automatic} \Rightarrow \texttt{USA}$. The resulting model induces 38 valid configurations. Even though the originally proposed constraint caused an unintended high variability reduction, it did not introduce any traditional anomalies, like dead or false-optional features [10]. Thus, such design flaws are difficult to detect using satisfiability-based analyses.

Reducing the variability of a product line may simplify quality assurance and maintenance [6, 38], as fewer products need to be considered. To achieve this effectively, it is necessary to know the impact of an edit regarding the variability reduction. However, it is difficult or even impossible to estimate the difference in the number of configurations without automated support. The cardinalities before and after multiple changes can be used as one of the criteria to select a possible change. Consider the following two changes to our running example: remove the gearbox type `Manual` or make `Radio` MANDATORY. These result in a remaining cardinality of 21 or 40, respectively. Thus, removing `Manual` is more effective regarding the variability reduction.

Thüm et al. [60] propose computing the added and removed configurations explicitly. While this provides the variability reduction implicitly, it is not feasible for large differences. In previous work, we showed that there are up to $\approx 10^{1500}$ added configurations after a new version of a feature model [59]. If we store the 10^{1500} configurations with a size of one byte each, we need 10^{1488} terabytes of memory. Enumerating and storing is not feasible but we can compute the change in the cardinality.

Rating Errors. Kübler et al. [38] propose using the cardinality of a feature model specialization for rating an error. A specialization is an adaptation of the feature model which adds additional constraints removing valid configurations [60]. The idea is that an erroneous subset of the configuration space included in more valid configurations is potentially more critical. Let FM_{ERR_1}, FM_{ERR_2} be two specializations of the feature model FM that are erroneous.

$$\begin{gathered}\#FM_{ERR_1} > \#FM_{ERR_2} \Rightarrow \\ FM_{ERR_1} \text{ is potentially more critical than } FM_{ERR_2}\end{gathered} \tag{6}$$

Suppose there are two specializations of FM that cause errors in the implementation of our car product line: FM' which always contains `Bluetooth` and `USB` and FM'' which always contains `Automatic` and at least one of `Radio` and `CD`. Those specializations induce 16 and 20 valid configurations, respectively. This can be used as an indicator to argue that the error induced by FM'' is more critical and that the developers should focus on this error.

Degree of Orthogonality. Feature-model analyses are typically complex because of sub-tree interdependencies specified by cross-tree constraints. If the impact of these interdependencies is small, the sub-trees can be evaluated separately to get a reasonable estimate. To analyze a single sub-tree, it is possible to use only local constraints [10, 19]. Czarnecki et al. [19] and Benavides et al. [10] define constraints as local if they are cross-tree constraints with only a single feature, or tree constraints. The degree of orthogonality describes the ratio between the cardinality of the feature model with all and with only local constraints [20]. A high degree of orthogonality indicates that interdependencies between subtrees have a low impact regarding the cardinality of the feature model. In this case, analyses can be performed locally [20]. Let FM be a feature model and FM_{local} a generalization that only considers the tree hierarchy and local cross-tree constraints. The degree of orthogonality is computed as follows:

$$DegreeOfOrthogonality(FM) = \frac{\#FM}{\#FM_{local}} \tag{7}$$

For our example, the degree of orthogonality is equal to the cross-tree variability factor (0.636) as there are no local cross-tree constraints. This indicates that analyses only considering local constraints may provide a reasonable estimate depending on the accuracy required for the application.

Configuration Relevance. We propose using the cardinality of a feature model to rate a configuration's relevance. For example, consider 1,000 products that were ordered from a product line. One product based on configuration C_1 was ordered 20 times. Without knowing the size of the configuration space, it is difficult to tell whether C_1 is more or less represented than an arbitrary configuration. The product line may contain 42 valid configurations like in our motivating example or even 10^{11} valid configurations like a smaller industrial model which we evaluated in previous work [59]. In the first case, C_1 has a below-average representation, but in the second case a highly above-average representation. Let L be the list of configurations, C_1 the configuration in question, and $|L_C|$ the

number of configurations equivalent to C in L.

$$RelevanceOfConfiguration(FM, L, C_1) = \#FM \cdot \frac{|L_{C_1}|}{|L|} \quad (8)$$

$RelevanceOfConfiguration(FM, L, C_1) = a$ states that C_1 appears a times as often in L compared to an arbitrary configuration. A relevance score a below one indicates, that C_1 has an below-average representation. A relevance score a higher than one indicates, that C_1 has an above-average representation which can be used to prioritize C_1 for testing and maintenance.

Cost Savings of a Product Line. Clements et al. [17] and Heradio et al. [28] use the cardinality of feature models to evaluate whether it is beneficial to develop a product family as product line (PL) or not (i.e., as standalone products) (NPL). The costs *CNPL* to develop standalone products can be estimated by multiplying *#FM* with a cost estimation *CP* to develop a single product.

$$CNPL(FM) = \#FM \cdot CP \quad (9)$$

For the cost of a product-line approach, the authors consider first building one standalone product [17, 28]. Afterwards, the common features (i.e., features that appear in more than one configuration) are changed to be reusable. The relative costs for developing the features for reusability are referenced by *CR*. A $CR = 2$ indicates that building the features for reuse is twice as much effort than developing it as a standalone product. Then, the rest of the products $\#FM - 1$ can be built with a reduced cost *CPR* [28]. The details on how to compute *CPR* are omitted in the following formula as they are not related to the cardinality of a feature model.

$$CPL(FM) = CR \cdot CP + (\#FM - 1) \cdot CPR \quad (10)$$

Overall, we are interested in the difference between the costs of implementing the family as a product line $CPL(FM)$ and the costs of all standalone products $CNPL(FM)$. This difference indicates the benefit of a product-line approach [17, 28]. Consider the following example where developing a standalone car costs $CP = 3$ and it costs twice as much to develop for reuse (i.e., $CR = 2$). Each additional car then costs $CPR = 2$ to be built. Overall, this results in $CNPL(FM) - CPL(FM) = (42 \cdot 3) - (2 \cdot 3 + 41 \cdot 2) = 39$ which indicates that it requires less effort to build all 42 possible cars with a product-line approach.

Summary. In this section, we presented six applications from the literature and three new proposals based on the cardinality of a feature model. After computing the cardinality of a feature model once, the result can be reused across all applications presented in this section. This can be used for deriving configurations, collecting statistical inferences from the feature model, estimate economical benefits, or to support developers. We examined the scalability of several #SAT solvers on computing cardinalities of feature models in our previous work and found that multiple solvers scale to a vast majority of industrial feature models [59].

5 CARDINALITY OF FEATURES

For several applications, it is important to only consider a subset of the configuration space, such as configurations that contain a certain feature. We propose the *cardinality of a feature* which we define as the number of valid configurations that contain that feature. A similar concept has been proposed with the *commonality of a feature* which describes the relative share of valid configurations that contain the feature [24, 49, 64]. In consequence, commonality is a value between zero and one that indicates the probability of the feature to appear in a given valid configuration [49].

Application	Source
Core, dead & false-optional	Trinidad et al. [64], Perez [50]
Atomic set candidates	**New**
Feature prioritization	**New**
CTC restrictiveness for features	**New**
Payoff threshold of a feature	Heradio et al. [28, 32]
Degree of reuse	Cohen et al. [18, 24, 28]
Homogeneity	Clements et al. [17], Fernandez et al. [24], Heradio et al. [28, 32]
Optimize configuration process	Chen et al. [16], Mazo et al. [41]

Table 2: Applications - Cardinality of Features

Definition 2 (Cardinality of a Feature). *Let $FM = (FEATS, CONST)$ be a feature model, $f \in FEATS$ a feature, and VC the set of valid configurations. The cardinality of f is defined as the number of valid configurations that contain f:*

$$\#FM_f = \left|\{C = (FM, I, E) \in VC \mid f \in I\}\right|$$

Definition 3 (Commonality of a Feature). *Let $\#FM$ and $\#FM_f$ be the cardinality of the feature model FM and a feature f, respectively. The commonality of f is defined as:*

$$Commonality(FM, f) = \begin{Bmatrix} \frac{\#FM_f}{\#FM} & \#FM \neq 0 \\ 0 & \#FM = 0 \end{Bmatrix}$$

Similar to the cardinality of a feature model, it is possible to compute the cardinality of a feature using a #SAT solver. Let FM be a feature model and F_{FM} a propositional formula with $FM \equiv F_{FM}$. The cardinality $\#FM_f$ of the feature f is equal to $\#(F_{FM} \wedge f)$ which can be used as input for a #SAT solver. In our running example, the feature *USB* appears in 20 out of the 42 valid configurations. Thus, its cardinality is 20 and its commonality is 0.476. Table 2 provides an overview for applications based on the cardinality of features.

Core, Dead & False-Optional Features. The commonality of a feature can be used to identify anomalies. Given a feature model FM, a feature f, and the commonalities of f and its parent p, the anomalies dead, core, and false-optional feature can be computed in the following way. If the entire feature model is void, we consider each feature to be dead as our definition of commonality implies.

$$Commonality(FM, f) = 1 \Leftrightarrow f \text{ is a core feature [50]} \quad (11)$$

$$Commonality(FM, f) = 0 \Leftrightarrow f \text{ is a dead feature [50, 64]} \quad (12)$$

$$Commonality(FM, p) = Commonality(FM, f) \text{ and } f \text{ is not mandatory} \Leftrightarrow f \text{ is false-optional} \quad (13)$$

We expect that existing analyses for anomalies that are based on satisfiability are less time-consuming. However, if the commonality of features is computed anyway, these defects can be identified without additional SAT calls. In our running example, `Carbody` has a cardinality of 42 which is equal to the cardinality of the feature model. Thus, its commonality is one, which indicates its core status.

Atomic Set Candidates. We propose to use the cardinality of features to find possible candidates for atomic sets. Features in the same atomic set always appear in the same number of valid configurations. Otherwise, there would be at least one configuration that contains only a subset of the features. However, two features

with the same cardinality may still be part of distinct valid configurations. Thus, they are not necessarily part of the same atomic set. The following formulas can be used to find atomic set candidates.

$$\#FM_f = \#FM_g \Rightarrow f, g \text{ are candidates for an atomic set} \quad (14)$$

$$\#FM_f \neq \#FM_g \Rightarrow f, g \text{ are not part of the same atomic set} \quad (15)$$

In our example, both `Navigation` and `GPSAntenna` appear in 24 valid configurations. Thus, they may be part of the same atomic set. A further analysis can examine whether the features are part of the same atomic set. `USB` appears in 32 valid configurations and, thus, is not part of this atomic set.

Feature Prioritization. We argue that the cardinality of a feature can be used as an indicator for the importance of this feature. For example, a developer may have to decide between two alternative features f_1, f_2 to implement next. To potentially create more distinct products, it is more effective to develop the feature with a higher cardinality first.

$$\begin{gathered}\#FM_{f_1} > \#FM_{f_2} \text{ and prioritize } f_1 \text{ over } f_2 \\ \Rightarrow \text{ Build more distinct products}\end{gathered} \quad (16)$$

In our running example, a product-line engineer may have to decide to implement `USB` or `CD` first. Those appear in 32 and 20 valid configurations, respectively. Therefore, it may be more beneficial to implement `USB` first.

In testing, it may be also interesting to prioritize features with a low cardinality as features with a high cardinality are covered with a high chance anyway. It is typically more likely to oversee a bug in a feature that appears in fewer valid configurations. Knowledge about feature's cardinalities can be used to lower the chances of missing an uncommon feature during testing. In our example, suppose the quality assurance tests five randomly configured cars. `Manual` is not part of the sample with a probability of 3.12%. `USA` has a 34.77% chance to not appear in the sample. Thus, prioritizing `USA` over `Manual` increases the odds for a wider coverage as `Manual` is more likely to be included in the sample anyways.

CTC Restrictiveness for Features. We argue that the cardinality of features can be used to identify features which are heavily restricted by cross-tree constraints. For this, we consider a feature model FM and an adaption $FM_{\overline{CTC}}$ without the cross-tree constraints. Then, we compare the cardinalities of a feature f in FM and in $FM_{\overline{CTC}}$. This information can be used to identify design errors.

$$CTCRestrictiveness(FM, f) = 1 - \frac{\#FM_f}{\#FM_{\overline{CTC},f}} \quad (17)$$

The *CTCRestrictiveness* lies between zero and one. Zero indicates that the feature is not restricted at all by cross-tree constraints. A value close to one indicates that the feature is heavily restricted through cross-tree constraints. Due to modernization of cars with `Automatic`, `CD` is only available for `Manual` cars. The developers introduce a new constraint `CD` $\Leftrightarrow$ `Manual` which unintentionally also requires every car with `Manual` to have a `CD`. These changes result in a $CTCRestrictiveness(\texttt{Manual}) = 1 - \frac{10}{66} = 0.85$ which indicates that `Manual` is heavily restricted. This information can be used to alert the developers to reconsider the added cross-tree constraint and change it to `CD` $\Rightarrow$ `Manual` instead. The adapted constraint results in $CTCRestrictiveness = 1 - \frac{21}{33} = 0.36$.

Payoff Threshold of a Feature. Typically, it is more expensive to develop a feature such that it can be reused for multiple products. However, the required adaptions to use the feature in other products is typically cheaper than to develop it from scratch. For a feature that appears only in a very small number of products it can be inefficient to engineer it for generic usage. The cardinality of a feature may indicate whether it is beneficial to develop it for reuse.

To achieve a return of investment, a feature that is developed for reuse must be reused a certain number of times. This break-even threshold depends on the added cost of developing a feature for reuse and the reduced cost of reusing it. If the number of valid configurations that contain a feature f is larger than the threshold, it is considered to be efficient to develop f for reuse [28, 32]. Let $Cost(f) = 1$ be the relative cost of implementing f from scratch. $CostToDevelopForReuse(f) = a$ indicates that its a times as expensive to develop f for reuse in the product line in contrast to develop it for standalone products with $Cost(f)$. After developing f for reuse, implementing f for a specific product only needs a proportion of the original cost. This reduced cost is described by $CostToReuse_f$.

$$PayoffThreshold(f) = \frac{CostToDevelopForReuse(f)}{1 - CostToReuse(f)} \quad (18)$$

$$\begin{gathered}\#FM_f < PayoffThreshold(f) \Rightarrow \\ \text{It is not efficient to develop } f \text{ for reuse.}\end{gathered} \quad (19)$$

Several authors proposed concrete procedures to compute the $CostToReuse(f)$ and $CostToDevelopForReuse(f)$ [11, 45]. However, the exact computations are beyond the scope of this work.

In our running example, suppose the cost of developing the feature `USA` for reuse is doubled. Furthermore, reusing `USA` costs 2/3 of the regular cost of implementing it. This results in a threshold of $\frac{2}{1-2/3} = 7$. Thus, `USA` needs to be reused at least seven times to break even. `USA` appears in eight valid configurations which indicates that it is beneficial to develop `USA` for reuse. The benefit potentially even increases further over time, as the configurable systems are typically growing [38, 59].

We expect that payoff thresholds only provide limited insights for industrial feature models. Boehm et al. [11] empirically evaluate the analysis but consider product lines with only a single digit number of valid configurations. Industrial feature models typically induce more than 10^{10} of valid configurations [38, 59].

Degree of Reuse. When implementing a product line, the degree of reuse can provide insight into the benefit of a product-line approach. The degree of reuse indicates the portion of products that is provided by common features (i.e., features that are part of at least two valid configurations). For example, a degree of reuse of 0.4 indicates that a typical product consists of 40% common features [18, 24, 28]. Eq. 20 quantifies the degree of reuse for the feature model FM with the set of common features $COM \subseteq FEATS$ [24, 28]. $Cost(f)$ indicates the effort to develop the feature f.

$$DegreeOfReuse(FM, COM) = \frac{\sum_{f \in COM}(Cost(f) \cdot \#FM_f)}{\sum_{f \in FEATS}(Cost(f) \cdot \#FM_f)} \quad (20)$$

Industrial feature models typically contain no features that appear in only one valid configuration as indicated by the data set used in our previous work [59]. In this case, the degree of reuse does not provide usable results as it is always one. The analysis is more suited for comparing a small number of products that only share some common features.

Suppose there exist three cars of the product family described by our running example that each contain a unique feature that is not part of the feature model. Every feature that directly appears in the feature model is part of at least two configurations and, thus, is common. For the sake of simplicity, we assume that every common feature has a cost of one. The three unique features have a cost of two. This leads to: $DegreeOfReuse = \frac{15}{15+6} = \frac{15}{21} = 0.71$. The result shows that the main effort to build the products comes from common features. This indicates a great benefit of product lines.

Homogeneity. The homogeneity of a product line describes the similarity of the valid configurations [17, 24, 28, 30, 32]. The homogeneity is a value between zero and one. A value close to zero indicates that the valid configurations are dissimilar (i.e., share a small number of features) while a value close to one indicates that the configurations are similar [24]. Clements et al. [17] propose to compute the homogeneity of a product line using the number of features that appear only in one valid configuration. However, Fernandez et al. [24] argue that this formula computes unexpected results for some cases in which highly heterogeneous configuration spaces are identified as homogeneous. Therefore, Fernandez et al. propose the commonality mean of all features [24].

$$Homogeneity(FM) = \frac{\sum_{f \in FEATS} Commonality(FM, f)}{|FEATS|} \quad (21)$$

With this formula, the homogeneity of our running example is 63% which indicates that the valid configurations have a high number of features in common.

Configuration Derivation Optimizations. Deriving a configuration can be improved with tool support, such as selection propagation [57]. Another option is ordering the features with different strategies to accelerate the derivation. First, a different feature order may reduce the number of configuration steps by performing a selection that implies a high number of other selections early. Second, assigning a feature that appears in many constraints may simplify the formula that is used for background analyses (e.g., selection propagation). Selecting such variables early in the configuration process may also accelerate the derivation.

Mazo et al. [41] propose six different heuristics to order features to accelerate the derivation. Two of the heuristics depend on #SAT. The first heuristic orders the features by their cardinality. The idea is to process features first that appear in a large variety of products [41]. The authors argue that this decreases the time required by the solver to find a valid configuration. However, we assume that this is neglected by the higher number of configuration steps required as the selection of a feature with high cardinality reduces the remaining possible selections by a smaller margin. The second heuristic prioritizes features that split the remaining configuration space in two partitions of similar size. Selecting such a feature halves the number of remaining valid configurations. Thus, following this procedure should reduce the number of required steps and the cost of analyses [41].

Chen et al. [16] propose to sort features by their selectivity to optimize product derivation. The selectivity of a feature depends on its cardinality. A highly selective feature has a high cardinality and implies the selection of a high number of other features which reduces the number of required configuration steps.

Application	Source
Uniform Random Sampling	Oh et al. [47, 48], Achlioptas et al. [2], Munoz et al. [44], Sharma et al. [55], Charkchakraborty et al. [15]
Atomic Sets	**New**
Rating Feature Interactions	**New**
Interactive Configuration Support	**New**

Table 3: Applications - Cardinality of Partial Configurations

Summary. In this section, we presented five applications from the literature and three own proposals based computing the cardinality or commonality for each feature of interest. The considered applications can be used for deriving configurations, collecting statistical inferences from the feature model, economic estimations, and support of developers.

6 CARDINALITY OF CONFIGURATIONS

For some applications, it is necessary to consider subsets of the configuration space that must contain multiple features and also allow to exclude some. We propose to compute the number of remaining valid configurations of a partial configuration for such applications which we define as the *cardinality of a partial configuration.* In addition, several applications considered for the cardinality of a feature can be extended for partial configurations. For example, instead of comparing the cardinality of a single feature in order to find priorities, a developer may also consider larger feature sets.

Definition 4 (Cardinality of a Partial Configuration). *Let $FM = (FEATS, CONST)$ be a feature model, $P = (I_P, E_P)$ a partial configuration, and VC the set of valid configurations. Then, the cardinality of P is defined as the number of valid configurations with all features included in P and without all features excluded in P:*
$\#FM_P = |\{C \in VC \mid I_P \subseteq I_C,\ E_P \subseteq E_C,\ I_P \cap E_C = \emptyset,\ E_P \cap I_C = \emptyset\}|$

The cardinality of a partial configuration can be computed by invoking a #SAT solver similarly as for a feature. For each included or excluded feature, we conjoin a corresponding positive or negative literal to the formula representing a feature model. The cardinality of a partial configuration $P = (FM, I, E)$ is $\#(F_{FM} \wedge (\bigwedge_{i \in I} i) \wedge (\bigwedge_{e \in E} \neg e))$. For our running example, there are four valid configurations where `Europe` and `GPSAntenna` are selected and `Manual` is deselected. Table 3 provides an overview of applications based on the cardinality of a partial configuration.

Uniform Random Sampling. While some anomalies can be found for an entire product line (e.g., dead features [10, 57]), a majority of operations for quality assurance, such as testing, only works on specific configurations [4, 42, 66]. Typically, these configurations are computed by sampling. Random configurations are useful as they can be used for representative testing, providing accurate statistical data, and evaluating other sampling methods [15, 48]. However, it is not trivial to create uniformly distributed random configurations. Randomly including or excluding features to create uniformly distributed configurations typically results in a vast majority of invalid configurations [46, 47, 59]. Thus, applying a filter to remove all invalid configurations is generally infeasible [47].

Uniform random sampling generates samples that guarantee true randomness of the included configurations. The idea is to create a one-to-one mapping between integers and the valid configurations of FM [44, 47]. Then, given a random number $r \in [1, \#FM]$, exactly one configuration can be selected. The resulting configurations should all be valid, uniformly distributed, and independent of the order variables are processed [44, 47].

Multiple authors [2, 15, 44, 47, 48, 55] propose algorithms for performing uniform random sampling. All of those rely on computing the cardinality of a partial configuration to decide about the inclusion or exclusion of features. The results of multiple empirical evaluations indicate that uniform random sampling scales for various industrial feature models [29, 48, 55].

Atomic Sets. In Section 5, we described how to use the cardinality of features to find candidates for atomic sets. Here, we extend this procedure to not only compute candidates but actual atomic sets by exploiting the following property. Consider two features f_1, f_2 that share the same commonality $\#FM_f$. If the number of valid configurations that contain both f_1 and f_2 is equal to $\#FM_f$, then f_1 and f_2 are part of the same atomic set. Let $V \subseteq FEATS$ be a set of features. The number of valid configurations that contain V is equivalent to $\#FM_{C_V}$ with $C_V = (FM, V, \emptyset)$.

$$f_i \in V : \#FM_{f_1} = \#FM_{f_2} = \ldots = \#FM_{f_n} = \#FM_{C_V} \Rightarrow \text{V is an atomic set.} \tag{22}$$

In our running example, `Navigation` and `GPSAntenna` appear in 24 valid configurations. Thus, they are potential candidates for an atomic set. Computing $\#FM_{C_V}$ with $V = \{$`Navigation`, `GPSAntenna`$\}$ also results in $\#FM_{C_V} = 24$ valid configurations. Thus, `Navigation` and `GPSAntenna` are part of the same atomic set. The features `Manual` and `Automatic` both are part of 21 valid configurations but $\#FM_{C_V}$ results in zero for $V = \{$`Manual`, `Automatic`$\}$. Therefore, the features are not part of the same atomic set.

Rating Feature Interactions for Sampling. We propose to use the cardinality of a partial configuration to rate feature interactions when sampling. Krieter et al. [37] propose a sampling algorithm that iteratively computes sets of configurations and then removes configurations with low scores. The authors propose to use the number of feature interactions that appear only in this configuration as the configuration's score. The goal is to reach a sample covering a high number of unique interactions after a number of iterations specified by the user. It may be beneficial to add the number of valid configurations that contain the interactions as a criterion for the configuration scores. Consider two configurations C_1, C_2 that both cover one interaction i_1, i_2 that does not appear in any other configuration of the sample. Suppose i_1 appears in more valid configurations. Then, i_1 is more likely to be covered by another configuration in the next iteration. Thus, C_2 should have a higher score than C_1 to increase the probability of covering more unique feature interactions.

$$\#FM_{i_1} > \#FM_{i_2} \Rightarrow score(i_1) < score(i_2) \tag{23}$$

In our running example, the interaction between `Radio` and `Manual` is in 20 of the 42 valid configurations. Thus, when creating samples for testing, it is very likely that this interaction is covered by some configuration of the sample. However, an interaction between `CD` and `DigitalCards` that appears in 8 valid configurations is less likely to be included in the sample. Thus, a configuration that contains the latter interaction should have a higher score.

Interactive Support for Derivation of Configurations. Deriving valid configurations for a feature model is complex because of the constraints imposed by the feature model. It is typically impossible for users to grasp the impact of a feature selection. We propose to interactively display the number of valid configurations for the current selection and the numbers that result from selecting unassigned features. For each currently unselected feature, the number of valid configurations that remain after the selection should be shown next to the feature. On one hand, this information can be used to grasp the impact of a feature selection. On the other hand, unexpected resulting numbers may help to find design flaws.

Summary. In this section, we presented one application from the literature and three own proposals based on the cardinality of a partial configuration opposed to single features or only once for the feature model. The considered applications can be used for sampling, deriving configurations, and to support developers.

7 CONCLUSION & FUTURE WORK

We conclude that applying #SAT technology to feature models has great potential, as we identified a variety of applications that are dependent on counting the number of valid configurations. Thus, applying #SAT for the analysis of feature models is beneficial for several aspects of the development process, such as economical estimations, guidance for developers, detecting design errors, sampling, and gathering statistical inferences. Each considered application depends on one of three metrics, namely the cardinalities of (1) a feature model (i.e., number of overall valid configurations), (2) features (i.e., number of valid configurations containing a specific feature), or (3) partial configurations (i.e., number of valid configurations that include/exclude the features specified in a configuration). Thus, it suffices to develop and optimize algorithms for these queries, to compute results for the 21 applications we considered.

In this work, we only considered naive invocations of #SAT solvers. We expect that optimizing the three considered queries yields performance improvements. Optimizations may exploit the similarity of multiple queries on the same feature model or reuse available information. Another important aspect for the usability of #SAT technology is its scalability. The empirical evaluation in our previous work [59] strongly indicated that computing the cardinality of feature models with off-the-shelf #SAT solvers scales to a vast majority of industrial feature models. The performance of computing the cardinality of features and partial configurations is currently largely unknown for industrial feature models. Therefore, an empirical evaluation on industrial feature models is paramount to make a strong conclusion about the usability of such analyses. There may be large product lines for which no off-the-shelf #SAT solver scales. In this case, evaluating the trade-off of approximate between accuracy and performance of #SAT solvers [3, 12, 27] may provide insights on their benefits.

ACKNOWLEDGMENTS

This work is based on the master's thesis of Sundermann [58].

REFERENCES

[1] Mathieu Acher, Philippe Collet, Philippe Lahire, and Robert B. France. 2011. Slicing Feature Models. In *Proc. Int'l Conf. on Automated Software Engineering (ASE)*. IEEE, Washington, DC, USA, 424–427. https://doi.org/10.1109/ASE.2011.6100089

[2] Dimitris Achlioptas, Zayd S Hammoudeh, and Panos Theodoropoulos. 2018. Fast Sampling of Perfectly Uniform Satisfying Assignments. In *Proc. Int'l Conf. on Theory and Applications of Satisfiability Testing (SAT)*. Springer, Berlin, Heidelberg, 135–147.

[3] Dimitris Achlioptas and Panos Theodoropoulos. 2017. Probabilistic Model Counting With Short XORs. In *Proc. Int'l Conf. on Theory and Applications of Satisfiability Testing (SAT)*. Springer, Berlin, Heidelberg, 3–19.

[4] Bestoun S. Ahmed, Kamal Z. Zamli, Wasif Afzal, and Miroslav Bures. 2017. Constrained Interaction Testing: A Systematic Literature Study. *IEEE Access* 5 (2017), 25706–25730. https://doi.org/10.1109/ACCESS.2017.2771562

[5] Sofia Ananieva. 2016. Explaining Defects and Identifying Dependencies in Interrelated Feature Models.

[6] Ebrahim Bagheri and Dragan Gasevic. 2011. Assessing the Maintainability of Software Product Line Feature Models Using Structural Metrics. *Software Quality Journal (SQJ)* 19, 3 (2011), 579–612.

[7] Ebrahim Bagheri, Tommaso Di Noia, Dragan Gasevic, and Azzurra Ragone. 2012. Formalizing Interactive Staged Feature Model Configuration. *J. Software: Evolution and Process* 24, 4 (2012), 375–400.

[8] Don Batory. 2005. Feature Models, Grammars, and Propositional Formulas. In *Proc. Int'l Systems and Software Product Line Conf. (SPLC)*. Springer, Berlin, Heidelberg, 7–20.

[9] David Benavides, Antonio Ruiz-Cortés, and Pablo Trinidad. 2005. Automated Reasoning on Feature Models. In *Proc. Int'l Conf. on Advanced Information Systems Engineering (CAiSE)*. Springer, Berlin, Heidelberg, 491–503.

[10] David Benavides, Sergio Segura, and Antonio Ruiz-Cortés. 2010. Automated Analysis of Feature Models 20 Years Later: A Literature Review. *Information Systems* 35, 6 (2010), 615–708.

[11] Barry Boehm, A Winsor Brown, Ray Madachy, and Ye Yang. 2004. A Software Product Line Life Cycle Cost Estimation Model. In *Proc. Int'l Symposium on Empirical Software Engineering (ISESE)*. IEEE, Washington, DC, USA, 156–164.

[12] Michele Boreale and Daniele Gorla. 2019. Approximate Model Counting, Sparse XOR Constraints and Minimum Distance. In *The Art of Modelling Computational Systems: A Journey from Logic and Concurrency to Security and Privacy*. Springer, Berlin, Heidelberg, 363–378.

[13] Jan Bosch, Gert Florijn, Danny Greefhorst, Juha Kuusela, J Henk Obbink, and Klaus Pohl. 2001. Variability Issues in Software Product Lines. In *Proc. Int'l Workshop on Software Product-Family Engineering (PFE)*. Springer, Berlin, Heidelberg, 13–21.

[14] Jan Burchard, Tobias Schubert, and Bernd Becker. 2015. Laissez-Faire Caching for Parallel# SAT Solving. In *Proc. Int'l Conf. on Theory and Applications of Satisfiability Testing (SAT)*. Springer, Berlin, Heidelberg, 46–61.

[15] Supratik Chakraborty, Daniel J Fremont, Kuldeep S Meel, Sanjit A Seshia, and Moshe Y Vardi. 2015. On Parallel Scalable Uniform SAT Witness Generation. In *Proc. Int'l Conf. on Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*. Springer, Berlin, Heidelberg, 304–319.

[16] Sheng Chen and Martin Erwig. 2011. Optimizing the Product Derivation Process. In *Proc. Int'l Systems and Software Product Line Conf. (SPLC)*. IEEE, Washington, DC, USA, 35–44. https://doi.org/10.1109/SPLC.2011.47

[17] Paul C Clements, John D McGregor, and Sholom G Cohen. 2005. *The Structured Intuitive Model for Product Line Economics (SIMPLE)*. Technical Report. Carnegie-Mellon University, USA.

[18] Sholom Cohen. 2003. *Predicting When Product Line Investment Pays*. Technical Report. Carnegie-Mellon University, USA.

[19] Krzysztof Czarnecki and Chang Hwan Peter Kim. 2005. Cardinality-Based Feature Modeling and Constraints: A Progress Report. In *Proc. Int'l Workshop on Software Factories (SF)*. ACM, San Diego, California, USA, 16–20.

[20] Krzysztof Czarnecki and Andrzej Wąsowski. 2007. Feature Diagrams and Logics: There and Back Again. In *Proc. Int'l Systems and Software Product Line Conf. (SPLC)*. IEEE, Washington, DC, USA, 23–34.

[21] Adnan Darwiche. 2004. New Advances in Compiling Cnf to Decomposable Negation Normal Form. In *Proc. European Conf. on Artificial Intelligence*. IOS press, Amsterdam, Netherlands, 318–322.

[22] Tom DeMarco. 1986. *Controlling Software Projects: Management, Measurement, and Estimates*. Prentice Hall PTR, USA.

[23] David Fernández-Amorós, Ruben Heradio Gil, and José Cerrada Somolinos. 2009. Inferring Information From Feature Diagrams to Product Line Economic Models. In *Proc. Int'l Systems and Software Product Line Conf. (SPLC)*. Springer, Berlin, Heidelberg, 41–50.

[24] David Fernández-Amorós, Ruben Heradio, José Antonio Cerrada, and Carlos Cerrada. 2014. A Scalable Approach to Exact Model and Commonality Counting for Extended Feature Models. *IEEE Trans. on Software Engineering (TSE)* 40, 9 (2014), 895–910.

[25] José A Galindo, Mathieu Acher, Juan Manuel Tirado, Cristian Vidal, Benoit Baudry, and David Benavides. 2016. Exploiting the Enumeration of All Feature Model Configurations: A New Perspective With Distributed Computing. In *Proc. Int'l Systems and Software Product Line Conf. (SPLC)*. ACM, New York, NY, USA, 74–78.

[26] Michael Godfrey and Qiang Tu. 2001. Growth, Evolution, and Structural Change in Open Source Software. In *Proc. Int'l Workshop on Principles of Software Evolution*. ACM, New York, NY, USA, 103–106.

[27] Carla P. Gomes, Ashish Sabharwal, and Bart Selman. 2006. Model Counting: A New Strategy for Obtaining Good Bounds. In *Proc. Conf. on Artificial Intelligence (AAAI)*. AAAI Press, 54–61.

[28] Ruben Heradio, David Fernández-Amorós, José Antonio Cerrada, and Ismael Abad. 2013. A Literature Review on Feature Diagram Product Counting and Its Usage in Software Product Line Economic Models. *Int'l J. Software Engineering and Knowledge Engineering (IJSEKE)* 23, 08 (2013), 1177–1204.

[29] Ruben Heradio, David Fernández-Amorós, José Antonio Galindo, and David Benavides. 2020. Uniform and Scalable SAT-Sampling for Configurable Systems. In *Proc. Int'l Systems and Software Product Line Conf. (SPLC)*. ACM, New York, NY, USA, 17:1–17:11.

[30] Ruben Heradio, David Fernández-Amorós, Christoph Mayr-Dorn, and Alexander Egyed. 2019. Supporting the statistical analysis of variability models. In *Proc. Int'l Conf. on Software Engineering (ICSE)*. IEEE, Montreal, Quebec, Canada, 843–853.

[31] Ruben Heradio, Hector Perez-Morago, David Fernández-Amorós, Roberto Bean, Francisco Javier Cabrerizo, Carlos Cerrada, and Enrique Herrera-Viedma. 2016. Binary Decision Diagram Algorithms to Perform Hard Analysis Operations on Variability Models. In *Proc. Int'l Conf. on Intelligent Software Methodologies, Tools and Techniques (SOMET)*. IOS Press, 139–154.

[32] Rubén Heradio-Gil, David Fernández-Amorós, José Antonio Cerrada, and Carlos Cerrada. 2011. Supporting Commonality-Based Analysis of Software Product Lines. *IET software* 5, 6 (2011), 496–509.

[33] Ayelet Israeli and Dror G. Feitelson. 2010. The Linux kernel as a case study in software evolution. *J. Systems and Software (JSS)* 83, 3 (2010), 485–501.

[34] David S Johnson. 1992. The NP-Completeness Column: An Ongoing Guide. *J. Algorithms* 13, 3 (1992), 502–524.

[35] Roberto J. Bayardo Jr. and Joseph Daniel Pehoushek. 2000. Counting Models Using Connected Components. In *Proc. Conf. on Artificial Intelligence (AAAI)*. AAAI Press / The MIT Press, 157–162.

[36] Sebastian Krieter, Reimar Schröter, Thomas Thüm, and Gunter Saake. 2016. *An Efficient Algorithm for Feature-Model Slicing*. Technical Report FIN-001-2016. University of Magdeburg, Germany.

[37] Sebastian Krieter, Thomas Thüm, Sandro Schulze, Gunter Saake, and Thomas Leich. 2020. YASA: Yet Another Sampling Algorithm. In *Proc. Int'l Working Conf. on Variability Modelling of Software-Intensive Systems (VaMoS)*. ACM, New York, NY, USA, Article 4, 10 pages. https://doi.org/10.1145/3377024.3377042

[38] Andreas Kübler, Christoph Zengler, and Wolfgang Küchlin. 2010. Model Counting in Product Configuration. In *Proc. Int'l Workshop on Logics for Component Configuration (LoCoCo)*. Open Publishing Association, Waterloo, Australia, 44–53. https://doi.org/10.4204/EPTCS.29.5

[39] Jean-Marie Lagniez and Pierre Marquis. 2017. An Improved Decision-DNNF Compiler. In *Proc. Int'l Joint Conf. on Artificial Intelligence (IJCAI)*. ijcai.org, 667–673.

[40] Rafael Lotufo, Steven She, Thorsten Berger, Krzysztof Czarnecki, and Andrzej Wąsowski. 2010. Evolution of the Linux Kernel Variability Model. In *Proc. Int'l Systems and Software Product Line Conf. (SPLC)*. Springer, Berlin, Heidelberg, 136–150.

[41] Raúl Mazo, Cosmin Dumitrescu, Camille Salinesi, and Daniel Diaz. 2014. Recommendation Heuristics for Improving Product Line Configuration Processes. In *Recommendation Systems in Software Engineering*. Springer, Berlin, Heidelberg, 511–537.

[42] Flávio Medeiros, Christian Kästner, Márcio Ribeiro, Rohit Gheyi, and Sven Apel. 2016. A Comparison of 10 Sampling Algorithms for Configurable Systems. In *Proc. Int'l Conf. on Software Engineering (ICSE)*. ACM, New York, NY, USA, 643–654. https://doi.org/10.1145/2884781.2884793

[43] Marcílio Mendonça, Andrzej Wąsowski, and Krzysztof Czarnecki. 2009. SAT-Based Analysis of Feature Models is Easy. In *Proc. Int'l Systems and Software Product Line Conf. (SPLC)*. Software Engineering Institute, Pittsburgh, PA, USA, 231–240.

[44] Daniel-Jesus Munoz, Jeho Oh, Mónica Pinto, Lidia Fuentes, and Don Batory. 2019. Uniform Random Sampling Product Configurations of Feature Models That Have Numerical Features. In *Proc. Int'l Systems and Software Product Line Conf. (SPLC)*. ACM, New York, NY, USA, 289–301. https://doi.org/10.1145/3336294.3336297

[45] Jarley Palmeira Nóbrega, Eduardo Santana de Almeida, and Sílvio Romero Lemos Meira. 2008. Income: Integrated Cost Model for Product Line Engineering. In *Proc. of Euromicro Conf. on Software Engineering and Advanced Applications*. IEEE, Washington, DC, USA, 27–34.

[46] Jeho Oh, Don Batory, Margaret Myers, and Norbert Siegmund. 2016. *Finding Product Line Configurations With High Performance by Random Sampling*. Technical Report. University of Texas at Austin, USA.

[47] Jeho Oh, Don Batory, Margaret Myers, and Norbert Siegmund. 2017. Finding Near-Optimal Configurations in Product Lines by Random Sampling. In *Proc. Int'l Symposium on Foundations of Software Engineering (FSE)*. ACM, 61–71. https://doi.org/10.1145/3106237.3106273

[48] Jeho Oh, Paul Gazzillo, Don Batory, Marijn Heule, and Maggie Myers. 2019. *Uniform Sampling from Kconfig Feature Models.* Technical Report TR-19-02. The University of Texas at Austin, Department of Computer Science.

[49] Joaquin Peña, Michael G Hinchey, and Antonio Ruiz-Cortés. 2006. Building the Core Architecture of a Multiagent System Product Line: With an example from a future NASA Mission. In *Proc. Int'l Workshop on Agent-Oriented Software Engineering (AOSE)*. Springer, Berlin, Heidelberg.

[50] Héctor José Pérez Morago. 2016. *BDD Algorithms to Perform Hard Analysis Operations on Variability Models.* Ph.D. Dissertation. Universidad Nacional de Educación a Distancia (UNED), Spain.

[51] Gilles Perrouin, Sagar Sen, Jacques Klein, Benoit Baudry, and Yves Le Traon. 2010. Automated and Scalable T-Wise Test Case Generation Strategies for Software Product Lines. In *Proc. Int'l Conf. on Software Testing, Verification and Validation (ICST)*. IEEE, Washington, DC, USA, 459–468. https://doi.org/10.1109/ICST.2010.43

[52] Richard Pohl, Kim Lauenroth, and Klaus Pohl. 2011. A Performance Comparison of Contemporary Algorithmic Approaches for Automated Analysis Operations on Feature Models. In *Proc. Int'l Conf. on Automated Software Engineering (ASE)*. IEEE, Washington, DC, USA, 313–322. https://doi.org/10.1109/ASE.2011.6100068

[53] Reimar Schröter, Sebastian Krieter, Thomas Thüm, Fabian Benduhn, and Gunter Saake. 2016. Feature-Model Interfaces: The Highway to Compositional Analyses of Highly-Configurable Systems. In *Proc. Int'l Conf. on Software Engineering (ICSE)*. ACM, New York, NY, USA, 667–678. https://doi.org/10.1145/2884781.2884823

[54] Sergio Segura. 2008. Automated Analysis of Feature Models Using Atomic Sets. In *Proc. Int'l Systems and Software Product Line Conf. (SPLC)*, Vol. 2. 201–207.

[55] Shubham Sharma, Rahul Gupta, Subhajit Roy, and Kuldeep S Meel. 2018. Knowledge Compilation Meets Uniform Sampling. In *Proc. Int'l Conf. on Logic for Programming, Artificial Intelligence, and Reasoning*. EasyChair, 620–636.

[56] Stefan Sobernig, Sven Apel, Sergiy Kolesnikov, and Norbert Siegmund. 2016. Quantifying Structural Attributes of System Decompositions in 28 Feature-Oriented Software Product Lines. *Empirical Software Engineering (EMSE)* 21, 4 (Aug. 2016), 1670–1705. https://doi.org/10.1007/s10664-014-9336-6

[57] Joshua Sprey, Chico Sundermann, Sebastian Krieter, Michael Nieke, Jacopo Mauro, Thomas Thüm, and Ina Schaefer. 2020. SMT-Based Variability Analyses in FeatureIDE. In *Proc. Int'l Working Conf. on Variability Modelling of Software-Intensive Systems (VaMoS)*. ACM, New York, NY, USA, Article 6, 9 pages. https://doi.org/10.1145/3377024.3377036

[58] Chico Sundermann. 2020. *Applications of #SAT Solvers on Product Lines.* Master's thesis. TU Braunschweig, Germany. https://doi.org/10.24355/dbbs.084-202009161329-0

[59] Chico Sundermann, Thomas Thüm, and Ina Schaefer. 2020. Evaluating #SAT Solvers on Industrial Feature Models. In *Proc. Int'l Working Conf. on Variability Modelling of Software-Intensive Systems (VaMoS)*. ACM, New York, NY, USA, Article 3, 9 pages. https://doi.org/10.1145/3377024.3377025

[60] Thomas Thüm, Don Batory, and Christian Kästner. 2009. Reasoning about Edits to Feature Models. In *Proc. Int'l Conf. on Software Engineering (ICSE)*. IEEE, Washington, DC, USA, 254–264. https://doi.org/10.1109/ICSE.2009.5070526

[61] Thomas Thüm, Christian Kästner, Sebastian Erdweg, and Norbert Siegmund. 2011. Abstract Features in Feature Modeling. In *Proc. Int'l Systems and Software Product Line Conf. (SPLC)*. IEEE, Washington, DC, USA, 191–200. https://doi.org/10.1109/SPLC.2011.53

[62] Marc Thurley. 2006. sharpSAT - Counting Models with Advanced Component Caching and Implicit BCP. In *Proc. Int'l Conf. on Theory and Applications of Satisfiability Testing (SAT)*. Springer, Berlin, Heidelberg, 424–429.

[63] Takahisa Toda and Takehide Soh. 2016. Implementing Efficient All Solutions SAT Solvers. *J. Experimental Algorithmics* 21 (2016), 1–12.

[64] Pablo Trinidad, David Benavides, and Antonio Ruiz Cortés. 2006. Isolated Features Detection in Feature Models. In *Proc. Int'l Conf. on Advanced Information Systems Engineering (CAiSE)*. Springer, Berlin, Heidelberg.

[65] Leslie G Valiant. 1979. The Complexity of Enumeration and Reliability Problems. *SIAM J. Computing* 8, 3 (1979), 410–421.

[66] Mahsa Varshosaz, Mustafa Al-Hajjaji, Thomas Thüm, Tobias Runge, Mohammad Reza Mousavi, and Ina Schaefer. 2018. A Classification of Product Sampling for Software Product Lines. In *Proc. Int'l Systems and Software Product Line Conf. (SPLC)*. ACM, New York, NY, USA, 1–13. https://doi.org/10.1145/3233027.3233035

C.2 AMAI'24: On the Benefits of Knowledge Compilation for Feature-Model Analyses

[A2C1] Chico Sundermann, Elias Kuiter, Tobias Heß, Heiko Raab, Sebastian Krieter, and Thomas Thüm. "On the Benefits of Knowledge Compilation for Feature-Model Analyses". In: *Annals of Mathematics and Artificial Intelligence (AMAI)* 92.5 (2024), pp. 1013–1050. DOI: 10.1007/s10472-023-09906-6

Published under Creative Commons Attribution 4.0 International (CC-BY), see https://creativecommons.org/licenses/by/4.0/.

Annals of Mathematics and Artificial Intelligence
https://doi.org/10.1007/s10472-023-09906-6

On the benefits of knowledge compilation for feature-model analyses

Chico Sundermann[1] · **Elias Kuiter**[2] · **Tobias Heß**[1] · **Heiko Raab**[1] · **Sebastian Krieter**[1] · **Thomas Thüm**[1]

Accepted: 17 October 2023
© The Author(s) 2023

Abstract
Feature models are commonly used to specify the valid configurations of product lines. As industrial feature models are typically complex, researchers and practitioners employ various automated analyses to study the configuration spaces. Many of these automated analyses require that numerous complex computations are executed on the same feature model, for example by querying a SAT or #SATsolver. With knowledge compilation, feature models can be compiled in a one-time effort to a target language that enables polynomial-time queries for otherwise more complex problems. In this work, we elaborate on the potential of employing knowledge compilation on feature models. First, we gather various feature-model analyses and study their computational complexity with regard to the underlying computational problem and the number of solver queries required for the respective analysis. Second, we collect knowledge-compilation target languages and map feature-model analyses to the languages that make the analysis tractable. Third, we empirically evaluate publicly available knowledge compilers to further inspect the potential benefits of knowledge-compilation target languages.

Keyword feature modeling, knowledge compilation, SAT, #SAT

Chico Sundermann
chico.sundermann@uni-ulm.de

Elias Kuiter
kuiter@ovgu.de

Tobias Heß
tobias.hess@uni-ulm.de

Heiko Raab
heiko.raab@uni-ulm.de

Sebastian Krieter
sebastian.krieter@uni-ulm.de

Thomas Thüm
thomas.thuem@uni-ulm.de

1 University of Ulm, Ulm, Germany

2 Otto-von-Guericke University, Magdeburg, Germany

Published online: 06 November 2023

1 Introduction

Product lines are commonly used to develop, test, and evolve a family of similar products to reduce the costs compared to developing each product separately [1–5]. Each product in a product line is composed of a set of reusable *features*, which are generally shared across multiple products [3, 6]. Typically, not every combination of selected features (i.e., a *configuration*) is valid and results in a functional product [7–9]. To specify the set of valid configurations for a given product line, engineers can use *feature models* [2, 10, 11]. A feature model consists of a set of features and a set of *constraints* limiting the valid configurations [12]. For instance, including one feature may require in- or excluding another feature.

Manually keeping track of all constraints is infeasible, as industrial feature models may contain thousands of features and hundreds of thousands of constraints [7, 13]. For example, the Linux kernel (as of November 2018) contains more than 14,000 features and 60,000 constraints [14]. Automated reasoning is typically used to analyze feature models, for example to check whether a given configuration is valid (i.e., it conforms to all constraints imposed by the feature model) [2].

Many feature-model analyses are computationally expensive [7, 9, 11, 15]. The de facto standard for analyzing feature models is to translate them into propositional logic and automate the analysis with tools, such as *SAT* [11] or *#SAT* [7, 9, 16] *solvers*. Feature models require the full expressiveness of propositional formulas [17], so checking whether a feature model has at least one valid configuration (i.e., a satisfying assignment) is NP-complete [11]. Analogously, computing the number of valid configurations is #P-complete [7, 9, 18]. Even further, many feature-model analyses depend on numerous complex solver queries [2, 7, 16]. For instance, existing tools compute for each feature whether it is core (i.e., it appears in every valid configuration) or dead (i.e., it appears in no valid configuration) to detect unintentional side effects in the modeling process [19, 20]. Naively employing a SAT solver to calculate such core and dead features requires $2 \cdot n$ SAT calls with n being the number of features in the worst case.

For computing feature-model analyses that require multiple solver queries, employing *knowledge compilation* seems promising [21, 22]. With knowledge compilation, an initial effort is taken to translate the propositional formula for a feature model into an artifact of a knowledge-compilation target language, such as a *binary decision diagram (BDD)* [23] or *deterministic decomposable negation normal form (d-DNNF)* [24]. In theory, such a knowledge-compilation artifact can then be used to perform repetitive computations more efficiently [25].

However, the research on leveraging knowledge compilation for feature-model analysis is still rather limited in terms of (a) considered analyses, (b) considered knowledge-compilation target languages, and (c) compiler scalability: Regarding feature-model analyses, it is well-known that some feature-model analyses require multiple complex queries [2, 9, 16, 26–28]. However, the insights on the underlying algorithmic problems are scattered over many works and extracting the complexity of the respective analysis often requires interpretation of the reader [2, 11, 16, 26, 29–33]. Regarding knowledge-compilation artifacts, only few knowledge-compilation target languages have been applied to compute a few feature-model analyses [22, 34]. However, many capabilities of well-known knowledge-compilation target languages [25] have not been utilized yet for feature-model analyses. For instance, while BDDs have been employed repeatedly [21, 35–37] and d-DNNFs found some adoption

in recent literature [22, 38–40], target languages that are popular in other domains, such as SDDs [41], have not been employed at all. Finally, regarding compiler scalability, knowledge compilation can only be sensibly applied for feature-model analyses if the compilation of feature models to the respective artifact scales appropriately. If the compilation takes too long or does not scale at all, the benefits (i.e., efficient queries) of the knowledge-compilation artifact vanish. However, besides preliminary work on some target languages [21, 22, 39, 40, 42], it is still largely unknown which artifacts can be compiled from feature models in reasonable time.

In this article, we aim towards closing these gaps by elaborating on the potential of applying knowledge compilation for feature-model analysis with the following contributions:

- **Classifying Feature-Model Analyses:** To give an overview on the complexity of feature-model analyses, we collect various analyses from the literature. For each identified analysis, we assess the complexity regarding the underlying computational problem (e.g., SAT or #SAT) and the number of solver queries potentially required in Section 4.
- **Identifying Knowledge-Compilation Target Languages:** To showcase the potential of knowledge compilation, we perform a literature survey to identify knowledge-compilation target languages and respective knowledge compilers. Further, we present a mapping between feature-model analyses and knowledge-compilation target languages, indicating which analyses are tractable (i.e., solvable in polynomial time) for which target language in Section 5.
- **Evaluating Compiler Scalability:** Finally, we analyze the scalability of different target languages by evaluating the identified knowledge compilers on 53 industrial feature models in Section 6.

2 Background & running example

A *product line* describes a family of products that share certain characteristics, called *features* [1–5]. Each product corresponds to a composition of features (a *configuration*) that appear in the product line. Typically, not every configuration leads to a useful product (i.e., it may contain errors or may be inefficient to produce). For instance, a car might be limited to have one type of gearbox, either automatic or manual.

Feature models are the de facto standard for specifying the valid configurations of a product line [2, 10, 11]. A feature model consists of a hierarchy of features (the *feature tree*) and additional propositional cross-tree constraints that further limit the set of valid configurations [11, 12]. In Fig. 1, we show an example feature model that describes the valid configurations of

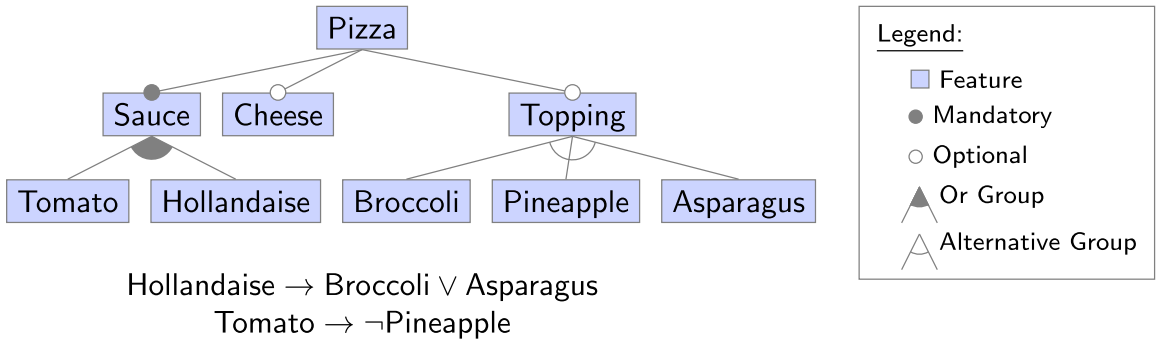


Fig. 1 Running example: Feature model representing pizza

a pizza product line. Each pizza requires a sauce indicated by the MANDATORY flag. As indicated by the OR group, a customer can select tomato or hollandaise as sauce (or both). A pizza may further include cheese as shown by the OPTIONAL flag. Finally, exactly one of three toppings (i.e., broccoli, pineapple, or asparagus) can be selected as part of an ALTERNATIVE group. In addition to the described feature tree, two cross-tree constraints below the feature tree, expressed as propositional formulas, specify further limitations: A customer cannot select the pineapple topping together with tomato sauce and needs to select either a broccoli or asparagus topping when selecting hollandaise sauce.

Formally, we define a feature model as a tuple $FM = (F, \Phi)$ over a set of features F and a set of constraints Φ, which includes both hierarchical and cross-tree constraints. A *configuration* $C = (I, E)$ consists of two sets $I, E \subseteq F$ containing included and excluded features, respectively. A feature cannot be included and excluded in the same configuration (i.e., $I \cap E = \emptyset$). A configuration that satisfies all constraints Φ imposed by the feature model is called *valid*. If all features are either selected or deselected (i.e., $I \cup E = F$), the configuration is called *complete*. Otherwise, it is called *partial* [2].

To employs standardized solvers for automated analyses, feature models are commonly translated to propositional logic [2, 7, 12, 43–45]. Every hierarchical constraint can be translated to semantically equivalent propositional logic [2, 11] by applying translation rules. Table 1 shows translations that are commonly employed in feature-model analysis [2, 19, 43, 44]. Note that multiple semantically equivalent translations for such constraints have been considered in the literature [2, 46]. In addition, the cross-tree constraints below the feature tree are typically expressed as propositional formulas [17]. Thus, without a loss of generality, we consider Φ to be a conjunction (as every constraint needs to be satisfied) of propositional formulas. The set of satisfying assignments of Φ is equivalent to the set of valid configurations for the feature model FM. As the analysis of satisfying configurations is known to be computationally complex, highly-optimized logic-based tools (i.e., *solvers*) can be invoked on Φ to reason about the configuration space described by the feature model FM [7, 9, 11, 12, 44].

Two well-known solver classes for feature-model reasoning are SAT [47, 48] and #SAT [49–51] solvers. A SAT solver is a tool that, given a propositional formula ϕ, determines whether ϕ is satisfiable or not. That is, $\text{SAT}(\phi) \equiv$ true when ϕ has at least one satisfying assignment and $\text{SAT}(\phi) \equiv$ false otherwise (i.e.ϕ contains some contradiction). A #SAT solver also takes as input a propositional formula ϕ, but determines *how many* satisfying assignments it has, thus generalizing the functionality of a SAT solver. That is, $\#SAT(\phi) \equiv n$ when ϕ has exactly n satisfying assignments. In Section 4, we give a detailed account of how SAT and #SAT solvers can be applied to analyze the configuration spaces described by a feature model.

Table 1 Translation of Feature Models into Propositional Formulas (**R**oot, **C**hild, **P**arent)

Modeling Construct	Propositional Formula	Intuition
Root feature	R	R is always required
Optional feature	$C \Rightarrow P$	C requires P
Mandatory feature	$C \Leftrightarrow P$	Optional + P requires C
Or group	$\bigvee_{1 \leq i \leq n} C_i \Leftrightarrow P$	Optional + P requires at least one C_i
Alternative group	$\bigvee_{1 \leq i \leq n} C_i \Leftrightarrow P \wedge$ $\bigwedge_{1 \leq i < j \leq n} \neg(C_i \wedge C_j)$	Optional + P requires exactly one C_i

CNF a specialization of *NNF*, is a common representation for propositional formulas. A formula is in *negation normal form (NNF)* when all its negations occur only in literals (so, conjunctions and disjunctions cannot be negated). Further, a formula is in *conjunctive normal form (CNF)* when it is a conjunction of clauses (i.e., disjunctions of literals) [52]. CNF in particular is highly relevant because both SAT and #SAT solvers typically require it as input format.

To solve arbitrary SAT and #SAT problems, exponential time is required in the worst case, as SAT is NP-complete [53] and #SAT is #P-complete [54]. The complexity of these problems can be a particular issue if numerous similar solver queries should be executed. A *query* is a computational operation on a given instance (e.g., a feature model) without adapting the instance [25]. To improve query performance, *knowledge-compilation target languages*, which typically enable faster querying than CNF, can be used. With *knowledge compilation*, a formula is compiled once into a *knowledge compilation artifact* (i.e., an instance of a knowledge-compilation target language), which can then be reused for numerous *tractable* (i.e., solvable within polynomial time complexity) queries, in the hope of reducing the overall required query time. In Section 5, we review knowledge-compilation target languages that are known to improve the asymptotic runtime of feature-model analyses (i.e., making them tractable).

3 Methodology

Our overall goal is to elaborate on the potential of applying knowledge-compilation to feature-model analysis. Thus, we aim to provide a reasonable representation of problem settings in feature-model analyses and the applicability of knowledge-compilation target languages for those problem settings. To this end, we performed (1) an expert survey to gather relevant feature-model analyses, (2) a literature survey to collect knowledge-compilation target languages and knowledge compilers (3), a GitHub search to identify further compilers. In the following, we describe our methodology used to extract information on feature-model analyses (results discussed in Section 4), knowledge-compilation target languages (results discussed in Section 5), and knowledge compilers (results discussed in Section 6).

Gathering Feature-Model Analyses

To motivate the relevance of knowledge compilation for feature-model analyses, we aim to showcase algorithmic problems relevant in the feature-modeling domain. We identify relevant problems in two phases: First, we collect feature-model analyses considered in the literature and applied in practice. Second, we categorize each analysis into the type of algorithmic problem (e.g., SAT or #SAT) that needs to be solved for the analysis. Further, we provide worst-case complexities with regard to the number of computations required for each analysis.

In the first phase, we performed an expert survey to identify articles based on our expertise in the feature-modeling domain. Hereby, we consider work that proposes analyses or already existing surveys on feature-model analysis [1, 2, 26, 55]. Each of the authors suggested a variety of articles to include in the survey. While our list of considered articles may fail to cover all available feature-model analyses, we expect that our list is a reasonable representation of analyses and in particular of underlying algorithmic problems in the literature.

In the second phase, we used the list of gathered analyses to derive an initial set of problem classifications. Afterwards, we performed a first categorization of each analysis

into the different problem classes. Then, we iteratively updated this categorization according to feedback of each author, also evaluating and eventually updating the initially proposed classifications. Note that while we aim to identify popular and efficient solutions, we may not identify the least complex solution for every considered feature-model analysis.

Gathering Knowledge-Compilation Target Languages & Knowledge Compilers

To accurately assess the potential of knowledge compilation in practice, we elaborate on the capabilities of knowledge-compilation target languages and scalability of publicly available knowledge compilers. To this end, we first conduct a literature survey to identify considered knowledge-compilation target languages and published knowledge compilers. Second, we perform a GitHub search for each of the identified knowledge-compilation target languages to collect further compilers. In Fig. 2, we provide an overview of the methodology for our survey.

In the first phase, depicted on the left side of Fig. 2, we start by gathering relevant literature. We perform a keyword search in common databases, namely *ACM*, *Springer*, and *IEEE*. Hereby, we search for papers including the term `"knowledge compilation"`. After collecting the set of papers, we filter according to our inclusion and exclusion criteria described in Table 2 to remove work not related to knowledge compilation of translating propositional formulas to target languages enabling more tractable queries. Hereby, we also exclude knowledge compilation of higher order logics (e.g., first-order logic) as we limit our elaborations on feature models representable as propositional logic (cf.Section 2).

After reducing the considered literature set, we aim to identify considered knowledge-compilation target languages and referenced knowledge compilers. In particular, we extract the data shown in Table 3. With D1–D4, we collect meta data about the submission. For D5, we collect all knowledge-compilation target languages mentioned in the paper, which we later use for our GitHub search. With D6, we record the availability of a compiler using the three categories (1) no compiler presented, (2) compiler is presented but not publicly available, and (3) compiler is publicly available. If available, we collect the URL to the considered compiler in D7. Lastly, we collect the input and target language of the compiler with D8 and D9, respectively.

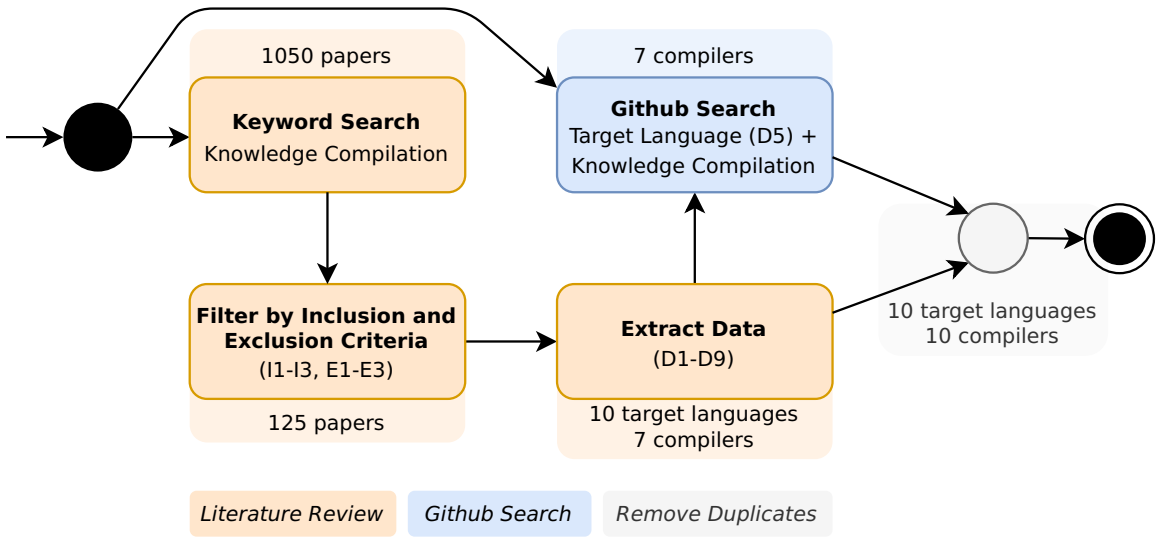


Fig. 2 Methodology for gathering knowledge-compilation target languages & knowledge compilers

Table 2 Inclusion (I) and Exclusion (E) Criteria

Index	Description
I1	Knowledge compilation is employed on propositional logic.
I2	Considered languages are tractable for queries relevant for feature-model analysis.
I3	Full text of the publication is available.
E1	Publication is not in English.
E2	Publication is not peer-reviewed.
E3	Publication is subsumed by another paper in the dataset.

We expect that some compilers have not been published in peer-reviewed work. Thus, after extracting the data in our literature survey, we perform a search on GitHub looking for further knowledge compilers. Hereby, we perform two searches on the GitHub database:

1. `"knowledge compilation" in:name,description,topics,readme`
2. For each target language considered in some paper of the survey (D5): `<target_language> in:name,description,topics,readme`

During the GitHub search, we include tools that accept dimacs (CNF) as input and compile to a considered target language. If the README and documentation do not provide sufficient information on how to build or run the tool, we exclude it. For BDDs, we use the knowledge compilers that we already identified in previous work with a similar procedure [21]. There, we collected hundreds of BDD libraries available on GitHub and extracted three popular tools which we use directly instead of performing another GitHub search.

By combining the set of knowledge compilers found in the survey and the GitHub search we aim to find a large variety of knowledge compilers. While we likely miss some available knowledge compilers, we expect to collect a reasonable representation of state-of-the-art knowledge compilers. With the collected compilers, we examine their scalability for feature-model analysis in our empirical evaluation (cf.Section 6).

Table 3 Data Extraction

Index	Data	Description
D1	Title	Title of Paper
D2	Authors	Authors of Paper
D3	Venue	Publishing Venue
D4	Year	Year of Publishing
D5	Target Languages	List of Knowledge-Compilation Target Languages
D6	Compiler Availability	Availability Score: (1) N/A, (2) Not Public, (3) Public
D7	Compiler URL	URL to Compiler if Publicly Available
D8	Input Language	Input Language for the Compiler (e.g., CNF)
D9	Target Language	Target Language of the Compiler (e.g., BDD)

4 Classifying feature-model analyses

Feature-model analyses depend on a variety of complex solving techniques, such as SAT [2] or #SAT [7, 9]. In practice, the majority of analyses are commonly computed by repetitively invoking a black-box solver on similar formulas [2, 10, 11, 26, 27, 29, 56–58]. In this section, we discuss various analyses to showcase the potential benefits of employing knowledge compilation. We present relevant solving techniques by classifying each analysis with regard to the algorithmic problem that it is typically reduced to. We also give an asymptotic assessment on the computational complexity of each analysis.

To assess the complexity of the gathered analyses, we elaborate on the number of queries that are required for an analysis on a given feature model. Hereby, we consider two dimensions: First, we provide the number of queries asymptotically required to compute a result for the corresponding analysis. Second, we differentiate between *static* analyses that are one-time computations on the feature model and *dynamic* analyses that can be applied numerous times with differing inputs, for instance in interactive settings. For example, while analyzing a partial configuration only requires one query, an interactive setting may require various analyses on different partial configurations but on the same unchanged feature model (dynamic). Other analyses just depend on the feature model which we consider as fixed and, thus, multiple computations do not yield additional results (static).

4.1 SAT-based analyses

In this section, we present different feature-model analyses that are typically based on SAT. In particular, given a propositional formula representing the feature model, possibly imposed with assumptions that (de-)select features, the satisfiability is checked to compute results for a given analysis. In the literature, a large variety of analyses have been proposed that are used for debugging, testing, sampling, and configuring [2, 27, 33, 59, 60]. Table 4 gives an overview of the collected feature-model analyses. The column *Queries* indicates the number of queries that are required for the analysis in the worst case (e.g., if implemented naively). *Repeatability* indicates whether an analysis is static or dynamic.

Consistency of Feature Models

For the first category, the analyses depend on the consistency of the entire feature model, which indicates whether at least one configuration is valid. The cross-tree constraints of a feature model may include a contradiction within each other or with tree constraints. In this case, we consider a feature model to be inconsistent. A consistent feature model is generally required for follow-up analyses and, thus, commonly checked first. Checking the consistency of a feature model typically corresponds to a single SAT query.

Definition 1 (Consistency of Feature Models) A given feature model $FM = (F, \Phi)$ is considered consistent if and only if its constraints in Φ do not contradict each other. Determining feature model consistency can be reduced to a satisfiability check as follows: $CO(FM) \equiv SAT(\Phi)$

The consistency of a feature model is used to check for a void [2] (i.e., inconsistent) feature model, which indicates a major design error. With a void feature model, it is impossible to derive products as every possible configuration of features is invalid. For a given

Table 4 Analyses based on consistency ($|F|$ = #Features, $|VC|$ = #Valid Configurations)

Type	Analysis	Queries	Repeatability
Cons. of Feature Models	Void Feature Model [2]	1	Static
Cons. of Features	Core Features [2]	$O(\|F\|)$	Static
	Dead Features [2]	$O(\|F\|)$	Static
	Variant Features [2]	$O(\|F\|)$	Static
Cons. of Partial Conf.	False-Optional Features [2]	$O(\|F\|)$	Static
	Atomic Sets [59]	$O(\|F\|^2)$	Static
	Valid Partial Configuration	1	Dynamic
	Decision Propagation [56]	$O(\|F\|^2)$	Dynamic
	Conditionally Core Features [2]	$O(\|F\|)$	Dynamic
	Conditionally Dead Features [2]	$O(\|F\|)$	Dynamic
	t-Wise Sampling [27]	$O(\|F\|^t)$	Static
	Non-Uniform Random Sampling	$O(1)$	Dynamic
	Most-Enabled-Disabled [60]	$O(1)$	Static
	One Enabled [60]	$O(\|F\|)$	Static
	One Disabled [60]	$O(\|F\|)$	Static
Cons. of Formulas	Redundant Constraint [2]	$O(\|\Phi\|)$	Static
	Feature-Model Edits [33]	$O(1)$	Static

feature model, a *single* consistency-query is sufficient for checking voidness. The feature model depicted in our running example is consistent. For instance, the configuration {Pizza, Sauce, Tomato} violates no constraint.

Consistency of Features

For many analyses, the consistency of the entire feature model is not sufficient. Some depend on the consistency of features, which indicates whether the feature is included in at least one valid configuration [2, 44]. Checking the consistency of numerous features typically requires solving multiple SAT problems, up-to the total number of features in a feature model in the worst case.

Definition 2 (Consistency of Feature) A given feature $f \in F$ is considered consistent in the feature model $FM = (F, \Phi)$ if and only if the feature f can be selected without violating any constraints in Φ. The consistency of a feature can be reduced to a satisfiability check as follows: $CO(FM_f) \equiv SAT(\Phi \wedge f)$.

The consistency of features is, for instance, used to identify *dead features* [2] which do not appear in any valid configuration. To detect all dead features, we need to solve a SAT problem for every single feature in the worst case [11]. Analogously, detecting *core* features (i.e., features that appear in every valid configuration) and *variant* features (i.e., features that appear in some, but not all valid configurations) requires a SAT call for each feature in the worst case, too. The core, dead, and variable features of a feature model correspond to the *backbone* of its formula, which can be computed efficiently in practice [61]; however, a linear number of SAT calls is still required in the worst case. Each status of a feature, namely

core, dead, and variant, is fixed for one version of a feature model. Hence, we consider each computation to be static.

In our running example, Sauce is a core feature as it is a mandatory child of the (always required) root feature. Thus, it is necessary to provide at least one type of sauce to sell pizza. Another example is Pineapple, which is a dead feature, possibly indicating a design error in the feature model.

Consistency of Partial Configurations

Various consistency-based analyses operate on partial configurations (i.e., a configuration that includes and excludes several, but not all features) [2]. A partial configuration is considered consistent if and only if there is at least one valid configuration that includes and excludes all features that are respectively included and excluded by the partial configuration. Analogously to consistency of features, computing the consistency of one partial configuration typically corresponds to a SAT call under assumptions (i.e., assigning literals a truth value). Depending on the analysis, it may be required to compute the consistency of various partial configurations.

Definition 3 (Consistency of Partial Configurations) A given partial configuration $C = (I, E)$ is considered consistent in the feature model $FM = (F, \Phi)$ if and only if the selection of all included features and deselection of all excluded features does not violate any constraints in Φ. The consistency of a partial configuration can be reduced to a satisfiability check as follows: $CO(FM_C) \equiv SAT(\Phi \wedge \bigwedge_{i \in I} i \wedge \bigwedge_{e \in E} \neg e)$.

The consistency of a partial configuration is regularly employed for analyses in the product-configuration process [2, 56]. For instance, checking whether the selection of a user can still lead to a valid complete configuration (i.e., is a *valid partial configuration*) and, thus, a usable product is often employed to implement configuration editors. This can be extended by *decision propagation*, which computes what features must (i.e., *conditionally core*) or cannot (i.e., *conditionally dead*) be selected anymore given a partial configuration. Decision propagation requires SAT calls up to the number of open features (i.e., not included or excluded in the partial configuration) for each partial configuration [56]. Typically, decision propagation is performed after every decision of the user (i.e., selection or deselection of a feature) over the remaining open features, thus requiring up to $\frac{|F|^2}{2}$ SAT calls to derive a complete configuration. As multiple configurations may be configured for a given feature model, decision propagation, valid partial configuration, conditionally core features, and conditionally dead features can be employed many times and, thus, we consider them to be dynamic analyses.

Several *t-wise sampling* algorithms depend on computing the consistency of a partial configuration [27, 32, 62–67]. Whenever a given t-wise interaction $T = \{a, b, ...\}$ is supposed to be covered by a configuration $C = (I, E)$ with $T \subseteq (I \cup E)$, the validity of this (partial) configuration must be ensured. Additionally, the validity of a t-wise interaction itself can be checked, before attempting to cover it, by interpreting it as a partial configuration. As one t-wise sample fully covers the t-wise interactions over a feature model, multiple computations for the same t are typically not beneficial. Thus, we consider t-wise sample as static analysis.

Non-uniform random sampling computes a list of configurations of a given size. To this end, each configuration is generated by finding one random satisfying assignment of the

feature-model formula. An algorithm may forbid the generation of duplicated configurations, which makes the problem of finding satisfying assignments progressively more difficult. As different random samples may intentionally be derived various times, we consider the non-uniform random sampling as dynamic.

Other sampling algorithms, such as *most-enabled-disabled*, *one-enabled*, and *one-disabled*, compute a fixed-size list of valid configurations with specific properties. Most-enabled-disabled computes exactly two configurations [60]. One with a minimal number of selected features and one with a maximal number of selected features. One-enabled computes one configuration per feature $f \in F$, which has the feature f selected and as many other features as possible deselected [60]. In contrast, one-disabled also creates one configuration per feature $f \in F$, but deselects the feature f and tries to included as many other features as possible [60]. We consider each of the three sampling approaches to be static as the respective samples are supposed to be stable for a given fixed feature model.

An *atomic set* describes features that behave as a unit (i.e., are always simultaneously included or excluded) [30]. In the worst case, the consistency of partial configurations corresponding to all pairs of features needs to be covered, leading to up to four SAT calls (i.e., $\Phi \wedge a \wedge b$, $\Phi \wedge a \wedge \neg b$, $\Phi \wedge \neg a \wedge b$, $\Phi \wedge \neg a \wedge \neg b$) for each of the pairs. To reduce this effort, other analysis results can be leveraged when computing atomic sets (e.g., the formula's backbone always corresponds to one atomic set). Computing atomic sets is static, as they remain equal for one version of a feature model.

Consistency of Formulas

Some analyses depend on arbitrary adaptations of the feature-model formula. While some analyses could be reduced to the consistency of features or partial configurations, applying assumptions to the original formula is often expressive enough.

Definition 4 (Consistency of Formula) For a given feature model $FM = (F, \Phi)$, the formula Φ' corresponds to an arbitrarily adapted formula $\Phi' = (\Phi \setminus \Phi_{removed}) \cup \Phi_{added}$ with Φ_{added} and $\Phi_{removed}$ being the set of added and removed clauses. The consistency of the formula Φ' can be reduced to a satisfiability check as follows: $CO(\Phi') \equiv SAT(\Phi')$.

When developing a feature model, some constraints may have no impact on the configuration space as their induced limitations are already expressed by other constraints [2]. Detecting such *redundant constraints* allows developers to clean their feature model to improve readability. Checking for one redundant constraint R can be performed by having a SAT call on $\Phi \setminus R \Rightarrow R$. To check all constraints for redundancy, up to $|\Phi|$ queries are required. As the constraints for a fixed feature model remain equal, we consider the analysis to be static.

There are many potentially insightful properties for comparing two feature models, such as differences in the set of features or set of cross-tree constraints [31]. While many of those depend on syntactical properties and can be easily extracted, other approaches allow insights on the semantic differences of two feature models [31, 33]. Thüm et al. [33] propose to classify edits to feature models into four types, namely generalization (i.e., added valid configurations), specialization (i.e., removed valid configurations), arbitrary edit (i.e., both added and removed), and refactoring (i.e., equal configuration space). To detect the edit type, $FM_1 \Rightarrow FM_2$ and $FM_2 \Rightarrow FM_1$ is checked.

4.2 #SAT-based analyses

In this section, we present analyses that are typically based on computing the number of valid configurations of a feature model. Computing the number of valid configurations enables a variety of analyses that can be used for debugging, development support, economical estimations, and statistical inferences [26]. A common method to compute the number of valid configurations is reducing the problem to #SAT [7, 9, 26, 55]. Table 5 gives an overview on the collected analyses based on #SAT indicating the complexity of each analysis w.r.t.the number of queries and repeatability.

Cardinality of Feature Models

Analogous to SAT-based analyses, the first type of #SAT-based analyses depends on analyzing the entire feature model [26]. The cardinality of feature models corresponds to the number of all valid configurations induced by the feature model. The cardinality of a feature model can be computed by invoking a #SAT solver on the formula representing the feature model.

Definition 5 (Cardinality of Feature Models) The cardinality $\#FM$ of a feature model $FM = (F, \Phi)$ is defined as the cardinality of the set of valid configurations $|VC|$. The feature-model cardinality can be reduced to computing a #SAT problem as follows: $\#FM \equiv \#\text{SAT}(\Phi)$.

Various analyses depend on the cardinality of a feature model [26]. The *variability factor* describes the share of valid configurations compared to all $2^{|F|}$ configurations [13]. *Variability*

Table 5 Analyses based on cardinality

Type	Analysis	Number of Queries	Repeatability
Card. of	Variability Factor [13]	1	Static
Feature Models	Variability Reduction [26]	1	Static
	Degree of Orthogonality [68]	1	Static
	Maintainability Prediction [69]	1	Static
	Configuration Relevance [26]	1	Static
	Cost Savings of Product Line [70]	1	Static
Card. of	Homogeneity [28]	$O(\|F\|)$	Static
Features	Atomic Set Candidates [26]	$O(\|F\|)$	Static
	Feature Prioritization [26]	$O(\|F\|)$	Static
	CTC Restrictiveness [26]	$O(\|F\|)$	Static
	Degree of Reuse [71]	$O(\|F\|)$	Static
	Optimize Configuring [72, 73]	$O(\|F\|)$	Static
	Payoff Threshold [55]	$O(\|F\|)$	Static
Card. of	Interactive Configuration [26]	$O(\|F\|^2)$	Dynamic
Partial Conf.	Uniform Random Sampling [16]	$O(\|F\|)$	Dynamic
	Rating Feature Interactions [26]	$O(\|F\|^2)$	Dynamic
Card. of	Rating Errors [9]	1	Dynamic
Formula	Variability of Feature Subsets [26]	1	Dynamic

reduction corresponds to using the change in cardinality after a new revision of the feature model to, for instance, detect errors [26]. Still, all analyses depend on a single query per feature model [26].

In our running example in Fig. 1, there are 14 valid configurations (i.e., 14 different pizzas that can be ordered). As there are overall $2^9 = 512$ theoretically possible configurations, the variability factor is $\frac{14}{512} \approx 2.73\%$.

Cardinality of Features

Other analyses depend on knowing the number of valid configurations that contain a certain feature (i.e., the cardinality of that feature). Computing the cardinality of a feature is typically reduced to invoking a #SAT solver on the formula representing the feature model with addition of an assumption for that feature.

Definition 6 (Cardinality of Features) The cardinality $\#FM_f$ of a feature $f \in F$ given a feature model $FM = (F, \Phi)$ is defined as the cardinality of the set of valid configurations that contain the feature f (i.e., $|\{C = (I, E) \mid C \in VC, f \in I\}|$). The feature cardinality can be reduced to computing a #SAT problem as follows: $\#FM_f \equiv \#\text{SAT}(\Phi \wedge f)$.

The cardinality of features can be used for economical estimations when considering to develop a family of products as product line [55, 71]. For instance, it may be beneficial to *prioritize features* that appear in many valid configurations to allow building more distinct products earlier in the development process [55]. Further, one may consider the *payoff threshold*, which indicates when the additional effort for building a feature for reuse in a product line is worth. A cardinality of the feature being higher than the payoff threshold may indicate that reusing the feature is beneficial. For both analyses, namely *feature prioritization* and *payoff threshold*, the cardinality for each feature of interest needs to be computed inducing linear effort in the worst case (i.e., $O(|F|)$).

Cardinality of features can also be employed to extract information about the variability of the feature model [26]. *Homogeneity* describes the similarity of valid configurations induced by a feature model [28]. The *degree of reuse* indicates the benefits of developing a software product line compared to standalone products by comparing the number of valid configurations induced by common features (i.e., features appearing in more than two valid configurations) and all features [71]. All features in an atomic set share the same cardinality. Hence, the cardinality of features can be used to identify *atomic set candidates* to reduce the number of interactions that need to be checked [26]. All three metrics listed above depend on the share of valid configurations each feature appears in and, thus, requires by definition the cardinality of each feature. *Cross-tree constraint (CTC) restrictiveness* describes how the cross-tree constraints impact the share of valid configurations a feature appears in compared to the limitations of the tree hierarchy [26]. As analyzing the cross-tree constraint restrictiveness may be beneficial for every feature, up to $|F|$ invocations may be employed.

Several strategies have been proposed for ordering features to select when configuring products [72, 73]. According to Mazo et al. [73], it may be reasonable to configure features with a high cardinality first as they tend to be more important in the product line. Selecting features with a small cardinality first may also be useful to accelerate the configuration process, as selecting these feature vastly reduces the number of remaining valid configuration and, thus, potentially the number of remaining decisions [72]. Fully sorting the features by their cardinality requires up to $|F|$ #SAT calls. Given a fix feature model revision, all analyses dependent on the cardinality of features are static as the cardinalities remain equal.

Cardinality of Partial Configurations

Further analyses are based on the number of valid configurations that include some and exclude other features (i.e., conform to a partial configuration) which we refer to as *cardinality of a partial configuration*. Note that the cardinality of partial configurations is a generalization of the cardinality of a features.

Definition 7 (Cardinality of Partial Configurations) The cardinality $\#FM_{C'}$ of a partial configuration $C' = (I', E')$ given a feature model $FM = (F, \Phi)$ is defined as the cardinality of the set of valid configurations that include all selected features and exclude all deselected features f (i.e., $\#FM_{C'} = |\{C = (I, E) \mid C \in VC, I' \subseteq I, E' \subseteq E\}|$). The cardinality of a configuration can be reduced to computing a #SAT problem as follows: $\#FM_{C'} \equiv \#\text{SAT}(\Phi \wedge \bigwedge_{i \in I} i \wedge \bigwedge_{e \in E} \neg e)$

The cardinality of partial configurations can be used for *uniform random sampling*, which computes a sample of configurations where each valid configuration has the same chance to appear in the sample [16, 26, 57]. While several algorithms have been proposed [22, 74], uniform random sampling can be reduced to #SAT [57]. Features are added to the configuration iteratively and for each feature the probability with which a feature needs to be included is computed. The probability is derived from the relative share of valid configurations the feature appears in based on the current partial configuration (i.e., the already included and excluded features). In the worst case, the probability for each feature (i.e., $O(|F|)$) needs to be computed requiring one #SAT call per feature. Depending on the use case, many configurations for a single feature model may be required and, thus, we consider the analysis to be dynamic.

While t-wise sampling is generally based on SAT (cf.Section 4.1), the cardinality of partial configurations can be used to potentially improve the sampling process. The runtime and sample size depends on the *prioritized feature interactions* for coverage. Explicitly prioritizing interactions that appear in few valid configurations may be beneficial as interactions with a high cardinality are more likely to be covered anyhow [26]. As interactions remain equal for a fixed feature model, we consider the analysis to be static.

When *interactively configuring* a product, it may be beneficial to indicate the consequences of a decision for the user. For instance, providing the number of remaining valid configurations after each decision may be helpful for the user [26]. This information can be extracted by computing the cardinality of partial configurations for each open (i.e., neither selected nor deselected) feature. Analogously to decision propagation, the computation needs to be done for all remaining features for each selection made and, thus, requires $O(|F|^2)$ #SAT calls. Further, many configurations can be derived, making the analysis dynamic.

Cardinality of Formulas

Analogously to SAT-based analyses, assumptions are not necessarily sufficient for #SAT-based analyses. Model counting on adapted formulas (e.g., by adding constraints or applying a projection) allows for further analyses.

Definition 8 (Cardinality of Formulas) For a given feature model $FM = (F, \Phi)$, the cardinality $\#\Phi'$ of a formula corresponds to the number of valid configurations induced by an arbitrarily adapted formula $\Phi' = (\Phi \setminus \Phi_{removed}) \cup \Phi_{added}$ with Φ_{added} and $\Phi_{removed}$ being the set of added and removed clauses. The cardinality of the formula Φ' can be reduced to a #SAT problem as follows: $\#\Phi' = \#\text{SAT}(\Phi')$.

The cardinality can be used to *rate errors* [9]. For instance, an error may appear if one of two features A and B is present. Adding a constraint $A \vee B$ to the formula and then invoking a #SAT solver results in the number of valid configurations the error appears in. An error that appears in many valid configurations may be more critical than another error that has a similar severity but appears in fewer valid configurations. Rating an error requires one #SAT call. However, for the same system there may be many errors and, thus, we consider the analysis to be dynamic.

In some cases, one is only interested in the *variability of a subset of features* [9]. For instance, only considering safety-critical features may be beneficial. Computing the variability for a feature subset can be done by performing a projection [75] (also called slice in the feature-modeling domain [76, 77]) and then invoking #SAT once. We further discuss projection in Section 4.4. Alternatively, projected model counting can be applied [75]. Note that projected model counting is considered to have a higher computational complexity than regular model counting [75, 78]. In particular, projected model counting is in $\#P^{NP}$ while model counting is #P-complete [75]. As potentially numerous feature subsets can be analyzed, we consider the analysis to be dynamic.

4.3 AllSAT-based analyses

For the third category, we present two analyses based on enumerating valid configurations induced by a feature model. Table 6 provides an overview on the analyses based on enumeration.

In some cases it may be beneficial to compute *all products* that can be derived from feature model [2]. The set of products (or valid configurations) could then be used to identify modeling errors or as test cases. As the set of valid configurations is equivalent to the set of satisfying assignments for the corresponding formula, computing all products can be reduced to solving one AllSAT problem. Often one is not interested in all valid configurations but a subset. A *filter* corresponds to deriving only valid configurations that conform to a partial configuration [2]. Computing all valid configurations for one filter corresponds to a single AllSAT call. As there are potentially numerous filters, the analysis is dynamic.

4.4 Algebraic-based analyses

Some feature-model analyses are also based on algebraic operations, such as building the conjunction of two feature models. Table 7 shows an overview on the feature-model analyses based on algebraic operations.

Feature-model slicing removes a set of features from a feature model without changing the dependencies between other features [76, 77]. Given a feature model $FM = (F, \Phi)$ and a feature set $F_S \subseteq F$ the slice of a feature model corresponds to $FM' = (F \setminus F_S, \Phi')$ such that the set of all satisfying assignments of Φ' is given by $A_{\Phi'} = \{a \setminus F_S \mid a \in A_\Phi\}$ where A_Φ is the set of satisfying assignments of Φ. Slicing a feature entails creating new

Table 6 Analyses based on AllSAT

Type	Analysis	Number of Queries	Repeatability
Model Enumeration	All Products [2]	1	Static
	Filter [2]	1	Dynamic

Table 7 Analyses based on algebraic operations

Type	Analysis	Number of Queries	Repeatability
Algebraic	Feature-Model Slicing [77]	1	Dynamic
	Feature-Model Composition [79]	1	Dynamic
	Feature-Model Differences [31]	1	Dynamic

clauses within the feature-model formula, which have to be checked for redundancy. The amount of new clauses is dependent on which feature is being sliced. In other domains, such a transformation for propositional formulas is also known as *existential quantification* [76], *projection* [75], and *forgetting* [25].

Feature-model composition corresponds to building a feature model representing the union of set valid configurations induced by two other feature models. The composition can be computed by a conjunction as follows: Given two feature models $FM_1 = (F_1, \Phi_1)$ and $FM_2 = (F_2, \Phi_2)$ the composition corresponds to $\Phi_1 \wedge \Phi_2$. While computing the composition for two feature models always computes equivalent results, a given feature can be compared to numerous feature models. Hence, we consider the analysis to be dynamic.

Feature-model differences compare two feature models with respect to the underlying formula Φ [31]. Acher et al. [31] compute a new feature model representing the differences in configuration spaces of the two considered feature models. To compute this feature model, Acher et al. [31] derive a formula which is satisfied by each assignment that satisfies exactly one of the corresponding feature models. Computing the feature-model differences depends on conjunction and negation of feature-model formulas [31]. For two feature models $FM_1 = (F_1, \Phi_1)$ and $FM_2 = (F_2, \Phi_2)$, the formula $(\Phi_1 \wedge \neg\Phi_2) \wedge (\Phi_2 \wedge \neg\neg\Phi_1)$ represents the valid configurations that appear in exactly one of the feature models. Analogously to feature-model composition, we consider the analysis to be dynamic as a fixed feature model can be compared to various feature models.

4.5 Summary

The potentially high number of queries on the same feature model motivates employing knowledge compilation. All 40 analyses presented above depend on computational complex problems (i.e., NP-complete or harder). 32 of the analyses potentially depend on multiple queries on the same feature model. Further, several analyses are often invoked multiple times which results in even more queries on the same formula.

Currently, it is not trivial for researchers and practitioners to select a promising knowledge-compilation target language. As the presented feature-model analyses are based on different computational problems (e.g., SAT or #SAT), the queries of interest vary depending on the use case. Thus, identifying a knowledge-compilation target language that both (1) supports fast queries of interest and (2) can be compiled efficiently, requires expertise on capabilities and performance of target languages.

5 Identifying knowledge-compilation target languages

In this section, we present all knowledge-compilation target languages we identified in our survey that are tractable for at least one feature-model analysis. A computational problem

is *tractable* for a given knowledge-compilation target language if and only if each artifact of that language enables computing a result for the given problem in polynomial time with respect to the size of the artifact [25].[1] We consider a feature-model analysis to be tractable for a target language if its underlying computational problem that needs to be solved is tractable for that target language. Also, we present insights on the succinctness which gives an indication on the size of a target language compared other languages. We envision that this knowledge on tractability and succinctness can guide researchers and practitioners to select suitable knowledge-compilation target languages depending on the feature-model analyses of interest. For each target language, we describe defining properties, provide simple examples, list the types of problems that are tractable, and provide insights on their succinctness.

5.1 Survey results

In our literature survey (cf. Section 3), we processed a total of 1,050 publications. After applying inclusion and exclusion criteria, 125 papers remained that address knowledge compilation of propositional formulas to knowledge-compilation target languages allowing tractable queries relevant for feature-model analysis.

Overall, we identified 10 target languages with overall 22 variants relevant for feature-model analysis (considering tractable queries). We use the term *variant* for minor specializations of a main target language. For instance, we consider s-d-DNNF as a d-DNNF variant as its still d-DNNF with one added property, namely smoothness [24]. In Table 8, we show the target languages we identified, their variants, and references that discuss these target languages in further detail. In this section, we limit the elaborations on the ten main target languages we identified and refer to the listed publications for details on the variants (cf.Table 8). For BDDs, we consider the variant *reduced ordered* BDDs (ROBDDs) as default if not stated otherwise which follows the literature on BDDs [23, 80]. Note that the list of references in the rightmost column does not include every work that mentions the target language; rather, we picked publications that provide evidence for our claims and discussions in the remainder of this section.

5.2 Knowledge-compilation target languages

In Table 9, we provide an overview on the tractability of each category of feature-model analysis we described in Section 4 for the identified knowledge-compilation target languages. A checkmark ✓ indicates the feature-model analysis is tractable for the respective knowledge-compilation target language. For instance, CNFs (which we include as a baseline for our comparison) are not tractable for any of the considered analyses, while MODS are tractable for all analyses. For CNF, DNF, DNNF, d-DNNF, PI, IP, BDD, and MODS the mapping to tractable feature-model analysis is based on the *knowledge-compilation map* provided by Darwiche and Marquis [25]. The remaining mappings are based on literature targeting the specific target language [81, 82, 89, 90].

Consistency of formulas (cf.Section 4.1) and cardinality of formulas (cf.Section 4.2) cannot be generally applied for knowledge compilation. Both analysis types, depend on arbitrarily changed formulas and, thus, not on a fixed feature-model instance. While some changes (e.g., adding an optional feature) may be applicable to a knowledge-compilation

[1] Note that AllSAT (cf.Section 4.3) is an exception as enumerating the satisfying assignments by definition requires at least linear time in the number of satisfying assignments. Hence, AllSAT is considered tractable if its runtime is polynomial with respect to the size of the artifact *and* the number of satisfying assignments [25].

Table 8 Literature survey result: Identified knowledge-compilation target languages

Target Language	Variants	References
Horn	Horn	[81]
EPCCL	EPCCL	[82]
DNF	DNF	[83]
DNNF	DNNF	[84, 85]
d-DNNF	d-DNNF	[86–88]
	Decision-DNNF	[86–88]
	s-d-DNNF	[85, 86, 88]
	s-Decision-DNNF	[85, 86, 88]
EADT	EADT	[89]
SDD	SDD	[41, 90]
	ZSDD	[91]
	Decision-SDD	[92]
BDD	BDD	[93]
	FBDD	[94]
	OBDD	[93]
	ROBDD	[93]
	ZBDD/ZDD	[95]
	C-OBDD	[96]
	C-NSOBDD	[96]
	DSDBDD	[97]
	TBDD	[98]
MODS	MODS	[25]

Table 9 Tractability of feature-model analyses for knowledge-compilation target languages

	SAT-Based			#SAT-Based			AllSAT	Algebraic		
	CO	CO_f	CO_C	$\#FM$	$\#FM_f$	$\#FM_c$	Enum.	Comp	Diff.	Slice
CNF										
Horn	✓	✓	✓							
EPCCL	✓	✓	✓							
DNF	✓	✓	✓				✓			✓
DNNF	✓	✓	✓				✓			✓
d-DNNF	✓	✓	✓	✓	✓	✓	✓			
EADT	✓	✓	✓	✓	✓	✓	✓			
SDD	✓	✓	✓	✓	✓	✓	✓			
PI	✓	✓	✓	✓	✓	✓	✓			✓
IP	✓	✓	✓	✓	✓	✓	✓			
ROBDD	✓	✓	✓	✓	✓	✓	✓	✓	✓	
MODS	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓

artifact in polynomial time, arbitrarily changed formulas may vastly differ from the original feature model. Hence, we decided to exclude consistency of formulas and cardinality of formulas in this section.

Besides tractability, succinctness is another often considered dimension for classifying knowledge-compilation target languages [25]. Informally, succinctness gives an indication on the expected size of an artifact in a given target language compared to other target languages. In combination with tractability, succinctness can be used to identify promising target languages for a given use case. The most succinct target language that is still tractable for the required analyses is often a promising option [25]. Formally, a target language A is at least as succinct as another language B (i.e., $A \leq B$) if and only if for every artifact in B there is a semantically equivalent artifact in A that is not exponentially larger than the original [25]. A is strictly more succinct than B iff $A \leq B$ but $B \not\leq A$.

Figure 3 shows the succinctness relations between the identified target languages. The figure is adapted from Darwiche and Marquis [25] and is extended by some additional languages, namely EPCCL, EADT, and SDD. An arrow $A \rightarrow B$ indicates that A is strictly more succinct than B. For instance, d-DNNF is strictly more succinct than SDD [90]. Note that the figure is not complete as some succinctness relations are unknown [25]. The references between the lines specify the publications that show the succinctness relation. The symbols below the respective target languages indicate which analyses are tractable. In Fig. 3, a

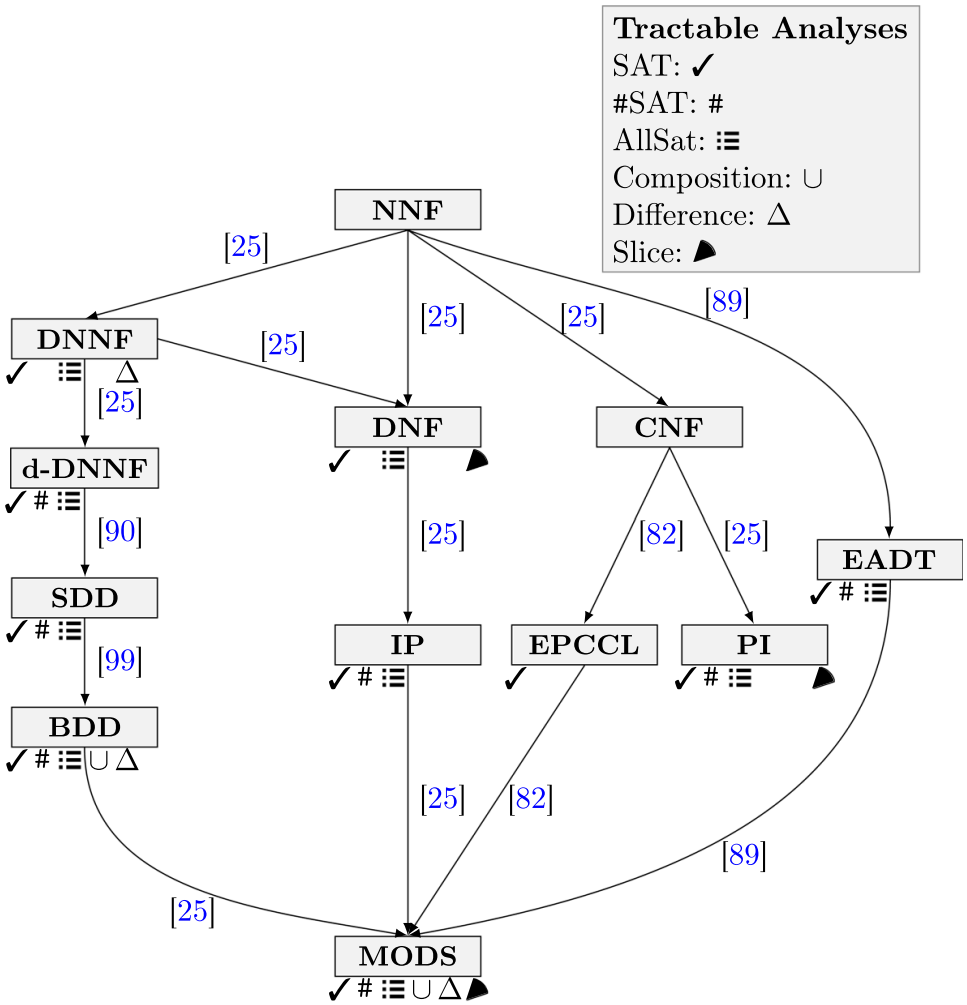


Fig. 3 Succinctness of languages

promising target language can be identified by the following characterization: the language supports all required queries, but its predecessors do not support all of them. Note that while succinctness is often a reasonable indicator, performance in practice may still vary. Hence, we also empirically evaluate publicly available compilers for different target languages in Section 6.

Horn Formula

A *Horn formula* is a CNF with an additional restriction, namely that each clause in a Horn formula contains at most one positive literal [81]. For instance, Eq. 1 is a Horn formula, as each of the three clauses contains either one or no positive literal. Checking the satisfiability of a Horn formula has linear time complexity with respect to the literals in the formula [81]. However, Horn formulas are not complete (i.e., they cannot represent every propositional formula) [25]. As feature models typically require the full expressiveness of propositional logic [17], Horn formulas are not applicable for all feature models.

$$(A \vee \neg B \vee \neg C) \wedge (\neg B \vee \neg D) \wedge (C \vee \neg E) \tag{1}$$

Each Pair Contains Complementary Literal

Each Pair Contains Complementary Literal (EPCCL) corresponds to a specialization of CNF where each pair of clauses contains complementary literals [82]. For a literal A, the literal $\neg A$ is complementary. Eq. 2 shows an example formula in EPCCL. Each pair of clauses contains complementary literals: $\{A, \neg A\}$, $\{B, \neg B\}$, and $\{\neg A, A\}$, respectively. EPCCL supports polynomial time queries for consistency of features models, features, and partial configurations [82].

$$(A \vee B \vee C) \wedge (\neg A \vee B \vee C) \wedge (A \vee \neg B \vee C) \tag{2}$$

Disjunctive Normal Form

A *disjunctive normal form (DNF)* is a disjunction of conjunctions where each of the conjunctions only consists of literals f or $\neg f$ [83]. For instance, Eq. 3 is in DNF. A DNF allows polytime queries for analyses based on consistency checking, model enumeration, and slicing [25].

$$(A \wedge \neg B) \vee (B \wedge C) \vee (A \wedge C \wedge D) \tag{3}$$

Decomposable Negation Normal Form

A *decomposable negation normal form (DNNF)* is a specialization of an NNF in which each conjunction is decomposable [25]. A conjunction is *decomposable* if and only if the conjuncts share no common variables. Figure 4 shows a directed acyclic graph representing the DNNF $(A \wedge B) \wedge (C \vee \neg C)$. Both conjunctions are decomposable as they share no variables (i.e., $\{A\} \cap \{B\} = \emptyset$ and $\{A, B\} \cap \{C\} = \emptyset$). Consistency checking, model enumeration, and slicing are tractable for DNNFs [25].

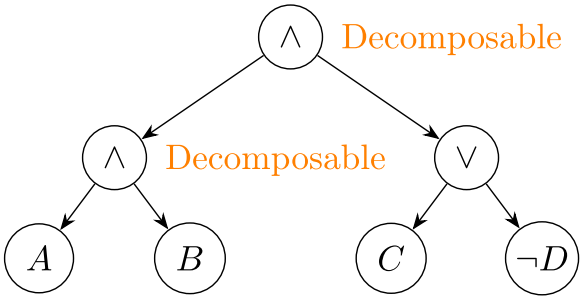


Fig. 4 Example formula in DNNF

Deterministic Decomposable Negation Normal Form

A *deterministic decomposable negation normal form (d-DNNF)* is a specialized NNF in which each disjunction is deterministic and every conjunction is decomposable [24, 25]. A disjunction is *deterministic* if and only if the disjuncts share no common solutions. Figure 5 shows a simple d-DNNF representing the formula $(A \vee \neg A) \wedge (B \vee \neg B)$ as a directed acyclic graph. The children of the conjunction ($\wedge$) share no variables (i.e., it is decomposable) and the children of each disjunction ($\vee$) share no solutions (i.e.they are deterministic). The combination of determinism and decomposability enables tractable queries for consistency checking, model counting, and enumeration [25].

Decision-DNNF, which is a subset of d-DNNF, are often applied [87, 88, 100]. Hereby, each disjunction is a decision node on a variable a with the structure $(\phi \wedge a) \vee (\tau \wedge \neg a)$ where ϕ and τ are subformulas that also conform to the properties of decision-DNNF. Due to its structure, each decision node is a deterministic disjunction by definition. Thus, all queries tractable for d-DNNF are tractable for decision-DNNF.

Smoothness is also often considered for d-DNNFs and decision-DNNFs [24, 25, 88]. A disjunction is *smooth* if all disjuncts of a disjunction share the same set of variables. While smoothness simplifies some queries, adding it does not enable more tractable queries for d-DNNF or decision-DNNF.

Extended Affine Decision Tree

An affine clause is a XOR (i.e., $\oplus$) over a set of literals. An *extended affine decision tree (EADT)* is a finite tree where each external node is a truth value (i.e., $\top$ or $\bot$) and the internal nodes are either affine decision nodes, affine decomposable conjunction, or affine decomposable disjunctions [89]. *Affine decision nodes* consists of an affine clause and two children corresponding to satisfying and not satisfying the affine clause, respectively. Alternatively, an affine decision node follows the structure: $((a_1 \oplus \ldots \oplus a_n) \wedge \Psi_l) \vee (\neg(a_1 \oplus \ldots \oplus a_n) \wedge \Psi_r)$.

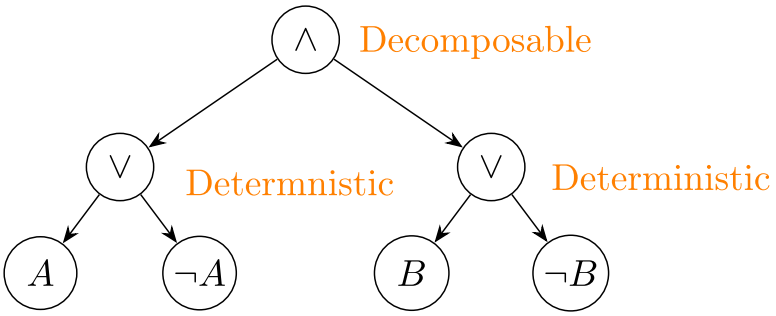


Fig. 5 Example formula in d-DNNF

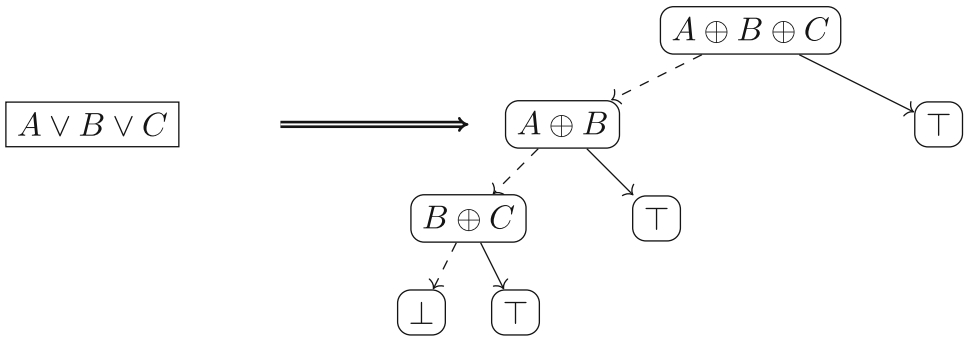


Fig. 6 Example formula in EADT

A conjunction is *affine decomposable* if and only if each of its conjuncts share no variables. Analogously, a disjunction is *affine decomposable* if and only if each of its disjuncts share no variables. Figure 6 shows an example EADT representing the formula $A \vee B \vee C$. The dashed edges indicate that its root term is unsatisfied while the solid line indicates that the root term is satisfied. Consistency checking, computing cardinalities, and model enumeration are tractable for EADTs.

Sentential Decision Diagram

Sentential decision diagram (SDD) is a generalization of BDD [41, 90]. Instead of having binary decision nodes on *one* variable (as in BDDs), decision nodes in SDDs may have arbitrarily many children on different disjunct sentential forms. More formally, an SDD is an (X, Y)-*partition* meaning the original formula is decomposed in the structure $(p_1(X_1) \wedge s_1(Y_1) \vee \ldots \vee (p_n(X_n) \wedge s_n(Y_n)))$ conforming additional properties. Hereby, $X_i \subseteq X$ and $Y_i \subseteq Y$ are variable sets with $X \cap Y = \emptyset$ and p_i (called *primes*) and s_i (called *subs*) are boolean functions over X_i and s_i, respectively. Such a decomposition is considered to be a (X, Y)-partition iff the primes are deterministic (i.e., share no solutions), exhaustive (i.e., $p_1 \vee \ldots \vee p_n = \top$), and satisfiable on their own (i.e., $\forall i : \text{SAT}(p_i) = \top$). Figure 7 shows an example SDD representing the formula $(A \wedge B) \vee (B \wedge C)$. The (X, Y)-partition decomposes the formula in $X = \{A, B\}$ and $Y = \{C\}$. The leaves show the sub formulas separated in *prime* | *sub*. SDDs enable polytime queries for consistency checking, cardinality computations, and model enumeration.

Prime Implicant

A term i (i.e., conjunction of literals) is an *implicant* of a formula F if and only if satisfying the term always satisfies F (i.e., $i \vdash F$). An implicant is *prime* if it cannot be further reduced (i.e., by removing literals) without losing the implicant property. The knowledge-compilation target language PI corresponds to a specialization of DNF where each term is a

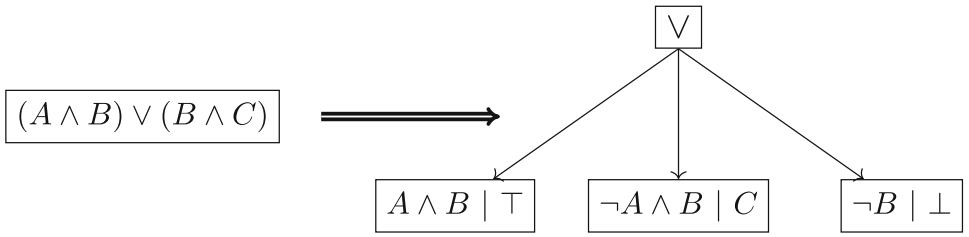


Fig. 7 Example formula SDD

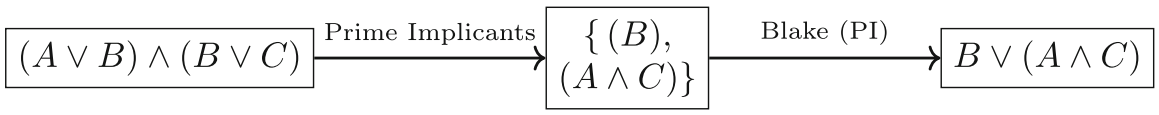


Fig. 8 Prime implicants for example formula

prime implicant [25]. A disjunction over all prime implicants (PI) of a formula F is also called the Blake canonical form of F. Figure 8 shows the prime implicants and PI representation for the formula $(A \vee B) \wedge (B \vee C)$. Consistency, equivalence, and model enumeration is tractable for prime implicants [25].

Prime Implicate

Prime implicates are the dual problem to prime implicants: A clause i (i.e., disjunction of literals) is an *implicate* of formula F if and only if satisfying the formula always satisfies i. An implicate is *prime* if it cannot be further reduced without losing the implicate property. The knowledge compilation target language IP corresponds to a specialization of CNF where each clause is a prime implicate [25]. Figure 9 shows the prime implicates and IP representation for the formula $(A \vee B) \wedge (B \vee C)$. Consistency, equivalence, and model enumeration is tractable for prime implicates [25].

Binary Decision Diagram

Binary Decision Diagrams (BDD) represent Boolean formulas as directed acyclic graphs with two terminal nodes $\boxed{0}$ and $\boxed{1}$. Every inner node in a BDD is associated to a variable of the formula and has precisely two outgoing edges (called high and low edge), denoting the assignment of true or false to this variable, respectively. Typically, one assumed BDDs to be *ordered* and *reduced* [93]. A BDD is ordered, when the sequence of variables on every path from the root node to the terminal nodes is the same. BDDs are reduced, when they neither contain a node with its outgoing edges incident with the same node (i.e., the assignment of the associated variable has no consequences on the outcome) nor two nodes associated to the same variable that have the same nodes incident to their high and low edges [93]. Figure 10 shows a BDD representing the formula $(A \wedge B) \vee (B \wedge C)$.

A plethora of feature-model analyses can be tractably performed on BDDs, namely consistency checking, cardinality computations, model enumerations, and some algebraic operations (i.e.diffing and composition).

Models

The class of formulas MODS (short for *models*) is the subset of formulas in DNF that satisfy determinism and smoothness (i.e.a kind of smooth d-DDNF) [25]. In essence, MODS enumerates all satisfying assignments, as each disjunct describes one satisfying assignment. Eq. 4 shows a propositional formula conforming to the restrictions of MODS. The formula

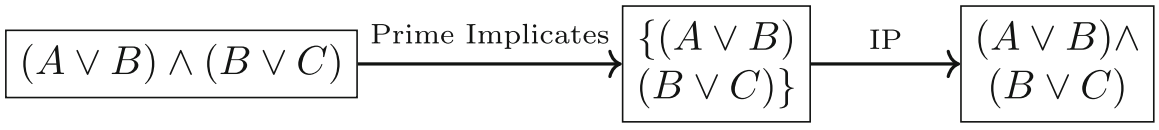


Fig. 9 Prime implicates for example formula

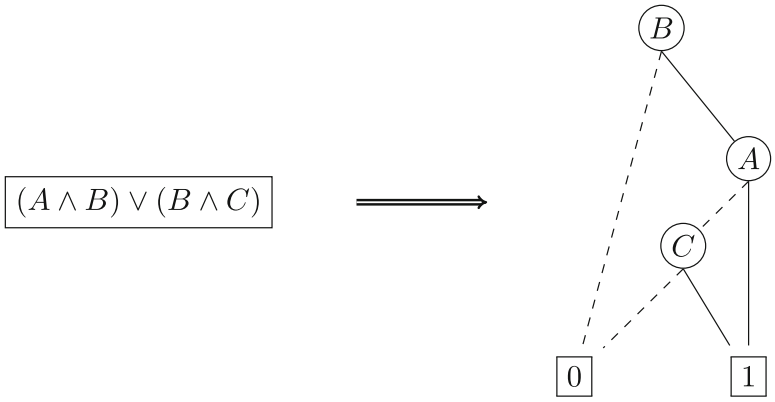


Fig. 10 Example formula BDD

is in DNF, the disjuncts share no solutions (deterministic), and contain the same variables (smooth). Each considered feature-model analysis is tractable given a MODS formula [25].

$$(A \wedge B \wedge C) \vee (A \wedge \neg B \wedge C) \vee (\neg A \wedge B \wedge C) \tag{4}$$

6 Evaluating compiler scalability

In addition to tractable queries, an important aspect for the applicability of knowledge compilation is the scalability of compiling to target languages. In our empirical evaluation, we examine the runtimes of publicly available knowledge compilers on compiling industrial feature models. The framework and input data used for the empirical evaluation is publicly available.[2]

6.1 Research questions

RQ1 How do knowledge compilers for different target languages perform on the task of compiling industrial feature models?
The benefits of knowledge compilation (i.e., fast online queries), are only applicable for feature-model analysis if the compilers scale to industrial feature models. To examine the applicability, we first inspect the general scalability regarding compilation time for different knowledge-compilation target languages. Second, we compare publicly available compilers to give recommendations on the best-performing tools for each target language.
RQ2 Which knowledge-compilation target languages should be used for the different types of feature-model analyses?
Typically, there is a tradeoff between tractability of queries and the complexity of compiling for selecting target languages. For each considered feature-model analysis, we aim to identify the most scalable target language that still allows polynomial queries for the respective analyses.

[2] https://github.com/SoftVarE-Group/kc-for-fmanalysis-evaluation/

Table 10 Subject systems

Domain	Models	Features	Clauses	Cardinality
Operating System	37	94–62,482	190–343,944	10^{10}–10^{417}†
Other Software*	13	11-31,012	1–102,705	10^{3}–10^{5}†
Automotive	2	384–18,616	1,020–350,221	10^{4}–10^{1534}
Financial Services	1	771	7,241	10^{13}

* Including archivers (2), database (2), video (2), network (1), garbage collection (1), image (1), compiler (1), solving (1), web development (1), and team management (1)
† Cardinalities are incomplete due to every solver hitting timeouts for some models

6.2 Subject systems

The applicability of a knowledge-compilation artifact is highly dependent on whether real-world instances can be compiled within a reasonable amount of time. Thus, we consider industrial feature models from a variety of domains that have been considered in other empirical evaluations.

Table 10 provides an overview of the considered systems sorted into application domains. First, we consider 39 feature models from the software systems and application domain provided by Oh et al. [16]. Second, we include three CDL[3], eight KConfig[4], and one automotive feature model published by Knüppel et al. [17]. Third, our dataset contains a BusyBox[5] feature model, which was extracted by Pett et al. [101]. Fourth, we consider a feature model from the financial service domain [102, 103]. Fifth, two additional models are available as FeatureIDE [19] examples.

6.3 Knowledge compilers

We include every found knowledge compiler that takes CNF (`dimacs`) as input and compiles to one of the knowledge-compilation target languages we identified. When collecting compilers, we identified seven compilers from the literature survey and seven compilers with the GitHub search resulting in overall ten knowledge-compilers.

Table 11 gives an overview of the identified knowledge compilers indicating the corresponding target language, the corresponding work, and a link to source files or binary. For each of d-DNNF, SDD, and BDD, our evaluation includes three compilers. We evaluate the BDD libraries `BuDDy` and `Cudd` by using the `dduerum` wrapper, we provided in previous work [21]. Further, we found one EADT compiler. We were not able to find compilers for other formats that accept CNF as input.

6.4 Experiment design

We invoke each listed knowledge compiler on every considered feature model. Thereby, we provide each feature model as CNF in `dimacs` format and perform ten repetitions. If a feature model was not already given as CNF, we translated it to `dimacs` using FeatureIDE

[3] http://ecos.sourceware.org/ecos/docs-latest/cdl-guide/language.html

[4] https://www.kernel.org/doc/html/latest/kbuild/kconfig-language.html

[5] https://github.com/PettTo/Measuring-Stability-of-Configuration-Sampling

Table 11 Evaluated knowledge compilers

Name	Artifact	Reference	Link
`c2d`	d-DNNF	[24]	[a]
`dSharp`	d-DNNF	[88]	[b]
`d4`	d-DNNF	[87]	[c]
`MiniC2D`	SDD	[104]	[d]
`SDD`	SDD	[41]	[e]
`ZSDD`	ZSDD	[91]	[f]
`Cudd`	BDD	[105]	[g]
`BuDDy`	BDD	[106]	[h]
`CNF2OBDD`	BDD	[107]	[i]
`CNF2EADT`	EADT	[89]	[j]

[a] http://reasoning.cs.ucla.edu/c2d/
[b] https://github.com/QuMuLab/dsharp
[c] https://github.com/crillab/d4
[d] http://reasoning.cs.ucla.edu/minic2d/
[e] http://reasoning.cs.ucla.edu/sdd/ [f] https://github.com/nsnmsak/zsdd
[g] https://github.com/vscosta/cudd
[h] http://buddy.sourceforge.net/manual/main.html
[i] http://www.sd.is.uec.ac.jp/toda/code/cnf2obdd.html
[j] http://www.cril.univ-artois.fr/KC/eadt.html

v3.8.3, which uses a translation based on equivalence transformations [14]. We set a timeout of 10 minutes for each knowledge compiler, as feature models are typically analyzed either interactively or in a CI/CD environment [19, 56, 108]. In both cases, runtimes longer than a few minutes typically disrupt the working flow. Further, increasing the timeout in preliminary experiments had no substantial impact on the results. We use the default parameters of every considered tool. For each measurement, we perform ten repetitions and collect the resulting knowledge-compilation artifact and runtime for further analysis.

Technical Setup The empirical evaluation was run on a designated server with the following specifications:

- Operating System: Ubuntu 20.04.4 LTS
- CPU: Intel(R) Xeon(R) CPU E5-260v3 with 2.40Ghz
- Overall RAM: 256 GB

During the measurements, no other major processes were performed on the machine. For each tool, we set the memory limit to 8 GB as we consider this a reasonable limit to expect for notebooks and PCs used in practice. Further, in preliminary experiments, we identified no substantial differences in measured runtimes when increasing the memory limit beyond 8 GB. For our empirical evaluation, we used a Python framework which internally uses the module `timeit` to measure runtimes.[6]

6.5 Results

Figure 11 shows the runtimes of all considered knowledge compilers on the 53 feature models. Each point on the x-axis corresponds to one feature model, which are sorted ascendingly by

[6] https://docs.python.org/3/library/timeit.html

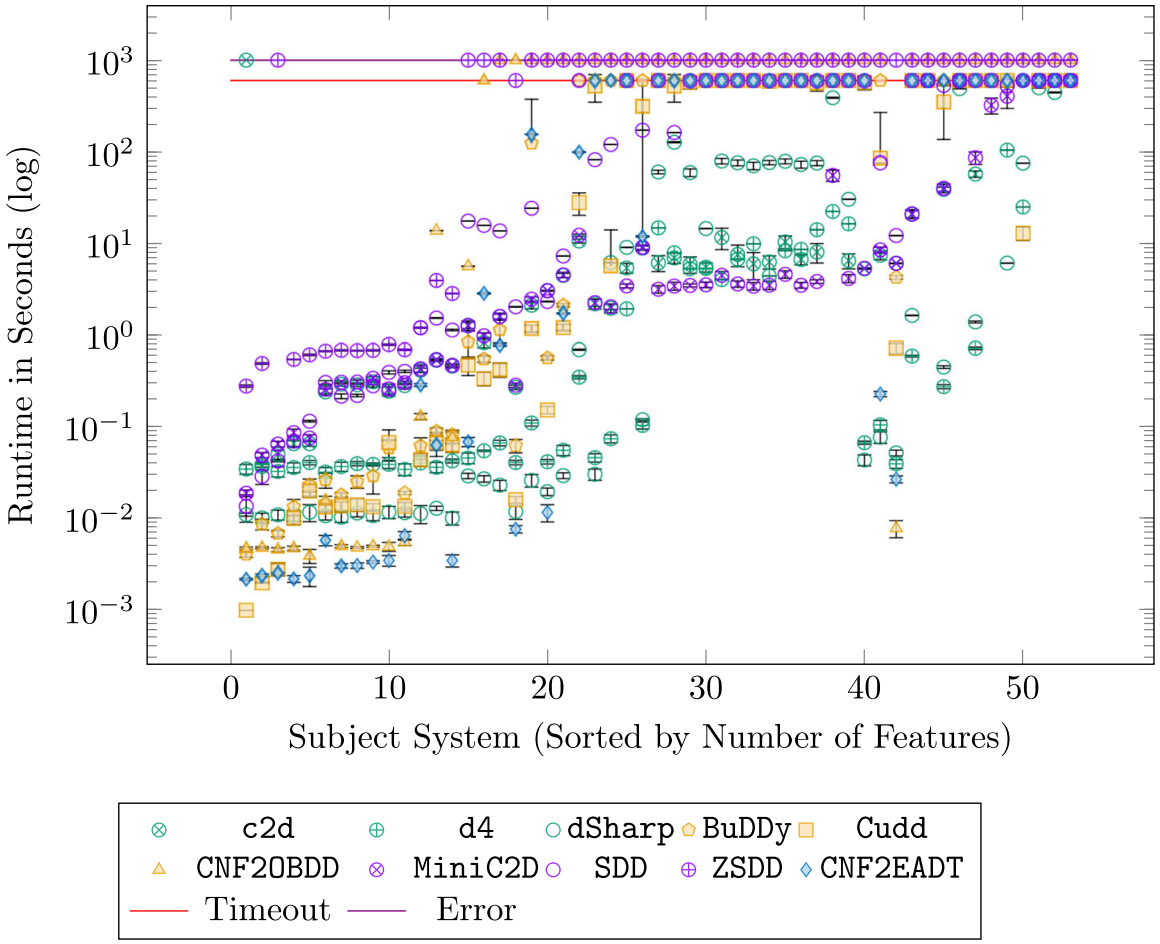


Fig. 11 Runtime knowledge compilers (average & standard deviation)

their number of features. The y-axis shows the runtime in seconds with a logarithmic scale. A mark on the red line indicates that the compiler hit the timeout for the respective feature model. The violet line indicates that a compiler terminated unexpectedly due to an error or hitting the memory limit.

Considering the different types of knowledge compilers, at least one of the d-DNNF compilers (`c2d`, `dSharp`, and `d4`) was able to compile 47 out of the 53 feature models. For the remaining six feature models, namely all (four) variants of *Linux*, *buildroot*, and *freetz*, all compilers for every target language hit the timeout or memory limit. At least one of the three SDD compilers, namely `MiniC2D`, `SDD`, and `ZSDD`, successfully compiled 46 feature models, failing to the same feature models as d-DNNFs and one additional feature model. The only EADT compiler `CNF2EADT` successfully compiled 25 feature models. The fastest BDD compiler `Cudd` was able to compile 28 feature models within the given timeout and memory limit.

Table 12 shows the number of successfully evaluated feature models and sum of runtimes for every evaluated knowledge compiler. The sum of runtimes corresponds to the runtimes over the feature models successfully evaluated by this compiler. The overall fastest d-DNNF compiler `d4` requires in sum 4.38 minutes for the 47 successfully evaluated feature models. For SDDs, the fastest compiler `MiniC2D` requires 19.7 minutes for the 46 feature models. `CNF2EADT` requires 4.54 minutes for the 25 feature models it successfully evaluated. `Cudd` successfully evaluated 28 of the feature models in overall 21.8 minutes.

Table 12 Overall runtime knowledge compilers

Compiler	Successes (of 53)	Runtime (Sum over Successes)
d4	47	4.38 Min.
dSharp	47	21.8 Min.
c2d	44	14.3 Min.
MiniC2D	46	19.7 Min.
SDD	29	29.5 Min.
ZSDD	13	14.1 Min.
CNF2OBDD	16	19.7 Min.
Cudd	28	21.8 Min.
BuDDy	22	2.23 Min.
CNF2EADT	25	4.54 Min.

6.6 Discussion

RQ1: How do knowledge compilers for different target languages perform on the task of compiling industrial feature models? The majority of evaluated feature models can be compiled to at least one of the knowledge-compilation target languages. Still, 6 out of 53 feature models could not be compiled to any target language within the given timeout of ten minutes. Preliminary measurements outside of the experiment design indicated that even substantially increasing the timeout (e.g., to one hour) leads to very similar results.

Compiling to d-DNNF and SDD is applicable for many industrial feature models (88.7% and 86.8%, respectively). Compiling to BDD or EADT is only applicable (w.r.t.the runtime) for less complex feature models. Besides one exception[7], the compilers for BDD or EADT only scale to feature models for which compiling to d-DNNF requires less than a second.

The compilers with the least overall runtime we identified for each target language are d4 (d-DNNF), MiniC2D (SDD), CNF2EADT (EADT, only one), and Cudd (BDD). For each target language, neither compiler is faster than all other compilers for that target language on all considered feature models. Thus, even though the compilers listed above seem a promising choice, it may still be beneficial to use different compilers depending on the input feature model.

Overall, our results indicate that knowledge compilation may be beneficial for feature-model analysis. The majority of feature models can be compiled within seconds to a target language that enables various tractable queries. To inspect the actual benefits, evaluating the performance of queries and comparing them with the current state of the art for feature-model analysis is essential as future work.

RQ2: Which knowledge-compilation target languages should be used for the different types of feature-model analyses? Based solely on the compilation time, *d-DNNFs* are the most promising target language for SAT-, #SAT-, and AllSAT-based analyses. SDDs are competitive with d-DNNFs for many feature models and, thus, are also a considerable option. The benefits of target languages also depend on the runtime required for queries and, thus, longer compilation times (e.g., to SDD) could potentially be amortized if the queries are faster. However, compilers for the other considered target languages (i.e., BDD

[7] The exception is Cudd successfully compiling the *embtoolkit* feature model.

and EADT) fail to compute a target artifact for many feature models within the timeout of ten minutes.

Employing SDDs might also be valuable in the future as more queries are tractable for SDD than for d-DNNF. In particular, equivalence and sentential entailment can be checked in polynomial time with SDDs but not with d-DNNFs [90]. SDDs are also tractable for additional transformations compared to d-DNNFs, such as negation and bounded conjunction/disjunction [90]. While the same set of the feature-model analyses we collected are tractable for d-DNNF and SDD, future analyses or alternative ways to solve existing analyses may capitalize on the additional tractability of SDD.

Compiling to BDD may also be beneficial due to additional tractable queries. In particular, feature-model analyses dependent on algebraic operations, namely feature-model diffing and feature-model composition, are tractable for BDDs but not for EADT, d-DNNF, nor SDD [25, 89, 90].

It is important to note that due to availability of compilers, our empirical evaluation only covers four out of the ten identified target languages. While Horn is not complete (i.e., cannot represent every propositional formula) [81] and, thus, not necessarily suitable for feature-model analysis, some of the six remaining target languages (cf.Section 5) may be beneficial [25, 82]. Developing compilers for promising (w.r.t.tractable queries) target languages may be beneficial future work for feature-model analysis and problems beyond the product-line domain.

6.7 Threats to validity

In this section, we discuss possible limitations to the results of our empirical evaluation and measures we have taken to reduce their impact.

Random Effects and Computational Bias Single measurements for the same combination of feature model and compiler may vary due to random effects and computational bias. For instance, compilation with `c2d` partially uses randomness to create the initial vtree which the d-DNNF is built upon [100]. To reduce the impact of such effects, we performed ten repetitions for each measurement.

Compiler Parameters Several considered compilers provide parameters (e.g., heuristics for selecting the branching variables) that potentially influence the compiler's runtime. Changing parameters may influence the performed measurements [109] and may even influence some conclusions. For such parameters, we always used the defaults as tweaking parameters would require further expertise. Hence, we expect default parameters to better reflect the usage of compilers for feature-model analysis in practice. Also, we consider parametrization of all compilers to be out of scope for this work. Still, we consider parameterizing the compilers as vital future work. A possible approach is to apply existing optimizers that more generically target parameter optimization for computationally complex problems [109, 110].

Formula Translations The translation from feature model to formula is not necessarily unique; for example, there are alternative encodings for expressing alternatives [46]. Also, the selection of a CNF translation, such as Tseitin's transformation [111] or equivalence transformations, may influence the performance of considered tools [14]. Integrating more feature-model or CNF translations for every feature model would vastly increase the complexity of the empirical evaluation. Thus, we consider different translations as out of scope for this work but as relevant in the future.

Further, some CNF translations do not ensure equivalence of formulas but less restrictive properties, such as equisatisfiability. For such an equisatisfiable formula, other analyses such as #SAT may product incorrect results [14]. Such a transformation may also result in faulty knowledge compilation artifacts. However, for each feature model not already given as CNF, we used the FeatureIDE transformation, which ensures equivalence.

External Validity: Feature Models Our empirical results cannot necessarily be transferred to other industrial feature models. However, we covered a wide range of feature models with regard to number of features (11–62,482), number of clauses (1–350,221), domains, and scalability of solvers and compilers (milliseconds vs. timeout for every compiler) [7].

External Validity: Compilers It is possible that we missed some available knowledge compilers that would influence the results and implications of our empirical evaluation. In particular, we only found compilers for four out of the eleven considered knowledge-compilation target languages. To effectively find available compilers, we systematically considered a large variety of research on knowledge compilation. We surveyed 1,050 papers including the term *knowledge compilation* and extracted various target languages and compilers. Afterwards, we extended the list of compilers further by performing a GitHub search for each identified target language. A possible explanation for missing compilers targeting specific languages may be their tractability, completeness, and succinctness. We could not identify compiler for the four least tractable languages (considering feature-model analyses), namely Horn [81], EPCCL [82], DNF [83], and DNNF [85]. Horn is also incomplete (i.e., cannot represent all propositional formulas) which may be another reason for the missing compiler. We also did not find a compiler for MODS which is the least succinct target language.

7 Related work

In this section, we discuss related work that (1) considers knowledge compilation in the feature-model domain and (2) surveys feature-model analyses. Further, we put our work into context of current research.

Knowledge Compilation on Feature Models

Binary decision diagrams have been considered for various feature-model analyses. Mendonca et al. [36] present several strategies for variable orderings based on the given feature model. Heradio et al. [35] use BDDs to compute core, dead, conditionally core, and conditionally dead features. In previous work [21], we analyzed the scalability of popular BDD libraries for compiling feature models. Our selection of BDD libraries in this work is based on the insights of the previous work [21]. Pohl et al. [37] compares the performance of several solving techniques, namely SAT solvers, BDDs, and CSP solvers for different feature-model analyses. Each of the listed publications considers only BDDs as target language and does not compare the scalability and capabilities (in terms of tractable queries) of BDDs to other target languages.

Sharma et al. [22] and Baital et al. [38] employ knowledge compilation to d-DNNF for uniform random sampling and t-wise sampling, respectively. However, their work is each limited to a single feature-model analysis and target language while we consider a large

variety of target languages and analyses. Bourhis et al. [40] perform several analyses on feature models using d-DNNFs. However, they do not consider any other target language. In previous publications, Kübler et al. [9] and we [7] used d-DNNF compilers as black-box #SAT solvers. In both publications, the knowledge-compilation artifact is not reused and, thus, the advantages of knowledge compilation are not used nor considered. Within another previous work [39], we employ d-DNNFs for model counting but consider no other analyses or target languages.

Voronov et al. [112] compare d-DNNFs (`c2d`) and BDDs (`BuDDy`) for enumerating valid configurations. Similar to our results, their empirical evaluation indicates that compiling to d-DNNF is substantially faster than compiling to BDD. In contrast to our work, the authors consider only one type of query (i.e., enumeration) and only two target languages. In previous work [42], we evaluate several model counters (or #SAT solvers) including some knowledge compilers capable of counting. Thereby, we exclude target languages intractable of counting and measure the runtimes required for counting instead of only compilation.

Krieter et al. [56] employ modal implication graphs to accelerate decision propagation (cf.Section 4.1). A modal implication graph encodes an underlying CNF as a directed graph indicating dependencies between features. Hereby, a binary clause $A \vee B$ is represented by two strong edges (i.e., $\neg A \Rightarrow B$ and $\neg B \Rightarrow A$) between the corresponding literals indicating the implications. Clauses with more than two literals (e.g. $A \vee B \vee C$) are represented by weak edges that indicate a possible implication (e.g., $\neg A \Rightarrow B \vee C$). While modal implications graphs reduce the number of required SAT calls and, thus, accelerate decision propagation in practice, none of the considered feature-model analysis is tractable for them. Krieter et al. [56] also consider an extension of modal implications graphs indicating all core and dead features. With this extension, feature model consistency and feature consistency are tractable for modal implication graphs. However, the set of core and dead features alone makes both analyses tractable and, thus, we do not consider pure modal implications graphs as a knowledge-compilation target language.

Surveys on Feature-Model Analyses

Benavides et al. [2] gathered various feature-model analyses depending on different computational problems (e.g., SAT, #SAT, or AllSAT). While giving general ideas on how to compute results for some specified analyses, the authors do not provide an assessment on the computational complexity regarding the number of queries. Further, we provide 25 additional analyses.

Galindo et al. [113] conduct a systematic literature survey focusing on publications dealing with automated analysis of feature models. However, they do not focus on concrete analyses but rather on publication metrics, such as popularity of venues and background of authors.

In previous work [26], we presented several analyses dependent on feature-model counting, fully subsuming the list of analyses of previous surveys on feature-model counting [9, 55]. There, we categorized each analysis by the type of computation, namely cardinality of feature model, features, and partial configurations. In contrast to this previous work, we now analyze the number of required queries and extend the list of analyses with analyses which depend on other computational problems.

In other previous work [1], we classified analysis strategies for product lines. This classification targets the application domain (e.g., product-based vs. family-based analysis) in contrast to showcasing underlying computational problems. Further, with the previous survey [1] we did not elaborate on concrete analyses but rather on more abstract analysis classes.

Various other surveys on product lines consider feature-model analyses to some degree [114–118]. While some publications provide rough ideas on the complexity of respective analyses, neither work systematically analyzes the complexities. In contrast, we provide a worst-case complexity for every collected feature-model analysis.

8 Conclusion

In our survey, we show that many feature-model analyses depend on numerous queries on the exact same feature model. Further, many analyses can be applied repeatedly for the same feature model, for instance when interactively deriving configurations. The potentially high number of required queries motivates the use of knowledge compilation. With knowledge compilation, the original feature model is translated to a target language that enables faster querying, potentially amortizing the initial cost of compilation.

Available target languages allow fast queries for different sets of analyses. Finding a sweet spot between tractable queries and compilation is essential to get full benefits from employing knowledge compilation. d-DNNFs and SDDs enable polytime queries for a variety of feature-model analyses and scale to the majority of considered industrial feature models. Hence, both target languages seem promising to apply. Other formats, such as BDDs and EADT, require substantially more time to compile, hitting the timeout of ten minutes for many feature models. Nevertheless, if a feature-model analysis of interest requires complex queries such as algebraic computations, BDD and EADT still yield benefits over d-DNNF and SDD.

9 Future work

Our results on the applicability of knowledge-compilation target languages and the scalability of available compilers indicate the potential of knowledge compilation. However, the benefits in practice also depend on the runtimes of querying the knowledge-compilation artifact. If the queries are too slow, the effort to compile the original formula might not amortize. To quantify the actual benefits, the efficiency of queries should be examined and compared with state-of-the-art solutions.

For many knowledge-compilation target languages we collected, we were not able to find any publicly available compilers. As each of the considered target languages enables some queries relevant for feature-model analysis, developing compilers may be beneficial.

Feature-model specific optimizations may improve the runtimes of knowledge compilers. In our work, we used the compilers as black boxes without any parameterization. Identifying and eventually automatically selecting parameters may increase the performance and, thus, could be valuable for future work.

Funding Open Access funding enabled and organized by Projekt DEAL.

Data Availability The datasets generated during and/or analyzed during the current study are available in the replication repository, https://github.com/SoftVarE-Group/kc-for-fmanalysis-evaluation/.

Declarations

Conflict of Interests The authors have no conflicts of interest to declare that are relevant to the content of this article.

Open Access This article is licensed under a Creative Commons Attribution 4.0 International License, which permits use, sharing, adaptation, distribution and reproduction in any medium or format, as long as you give appropriate credit to the original author(s) and the source, provide a link to the Creative Commons licence, and indicate if changes were made. The images or other third party material in this article are included in the article's Creative Commons licence, unless indicated otherwise in a credit line to the material. If material is not included in the article's Creative Commons licence and your intended use is not permitted by statutory regulation or exceeds the permitted use, you will need to obtain permission directly from the copyright holder. To view a copy of this licence, visit http://creativecommons.org/licenses/by/4.0/.

References

1. Thüm, T., Apel, S., Kästner, C., Schaefer, I., Saake, G.: A classification and survey of analysis strategies for software product lines. ACM Computing Surveys (CSUR) **47**(1), 6–1645 (2014). https://doi.org/10.1145/2580950
2. Benavides, D., Segura, S., Ruiz-Cortés, A.: Automated analysis of feature models 20 years later: A literature review. Information Systems **35**(6), 615–708 (2010)
3. Pohl, K., Böckle, G., van der Linden, F.J.: Software Product Line Engineering: Foundations, Principles and Techniques. Springer, Berlin, Heidelberg, Germany (2005). https://doi.org/10.1007/3-540-28901-1
4. Berger, T., Nair, D., Rublack, R., Atlee, J.M., Czarnecki, K., Wąsowski, A.: Three Cases of Feature-Based Variability Modeling in Industry. In: Proc. Int'l Conf. on Model Driven Engineering Languages and Systems (MODELS), pp. 302–319. Springer, Berlin, Heidelberg, Germany (2014). https://doi.org/10.1007/978-3-319-11653-2_19
5. McGregor, J.: Testing a Software Product Line. In: Testing Techniques in Software Engineering, pp. 104–140. Springer, Berlin, Heidelberg, Germany (2010)
6. Apel, S., Batory, D., Kästner, C., Saake, G.: Feature-Oriented Software Product Lines. Springer, Berlin, Heidelberg, Germany (2013). https://doi.org/10.1007/978-3-642-37521-7
7. Sundermann, C., Thüm, T., Schaefer, I.: Evaluating #SAT Solvers on Industrial Feature Models. In: Proc. Int'l Working Conf. on Variability Modelling of Software-Intensive Systems (VaMoS). ACM, New York, NY, USA (2020). https://doi.org/10.1145/3377024.3377025
8. Lotufo, R., She, S., Berger, T., Czarnecki, K., Wąsowski, A.: Evolution of the Linux Kernel Variability Model. In: Proc. Int'l Systems and Software Product Line Conf. (SPLC), pp. 136–150. Springer, Berlin, Heidelberg, Germany (2010)
9. Kübler, A., Zengler, C., Küchlin, W.: Model Counting in Product Configuration. In: Proc. Int'l Workshop on Logics for Component Configuration (LoCoCo), pp. 44–53. Open Publishing Association, Waterloo, Australia (2010). https://doi.org/10.4204/EPTCS.29.5
10. Sprey, J., Sundermann, C., Krieter, S., Nieke, M., Mauro, J., Thüm, T., Schaefer, I.: SMT-Based Variability Analyses in FeatureIDE. In: Proc. Int'l Working Conf. on Variability Modelling of Software-Intensive Systems (VaMoS). ACM, New York, NY, USA (2020). https://doi.org/10.1145/3377024.3377036
11. Mendonça, M., Wąsowski, A., Czarnecki, K.: SAT-Based Analysis of Feature Models is Easy. In: Proc. Int'l Systems and Software Product Line Conf. (SPLC), pp. 231–240. Software Engineering Institute, Pittsburgh, PA, USA (2009)
12. Batory, D.: Feature Models, Grammars, and Propositional Formulas. In: Proc. Int'l Systems and Software Product Line Conf. (SPLC), pp. 7–20. Springer, Berlin, Heidelberg, Germany (2005). https://doi.org/10.1007/11554844_3
13. Benavides, D., Trinidad, P., Ruiz-Cortés, A.: Using Constraint Programming to Reason on Feature Models. In: Proc. Int'l Conf. on Software Engineering and Knowledge Engineering (SEKE), pp. 677–682 (2005)
14. Kuiter, E., Krieter, S., Sundermann, C., Thüm, T., Saake, G.: Tseitin or not Tseitin? The Impact of CNF Transformations on Feature-Model Analyses. In: Proc. Int'l Conf. on Automated Software Engineering (ASE), pp. 110–111013. ACM, New York, NY, USA (2022). https://doi.org/10.1145/3551349.3556938
15. Oh, J., Gazzillo, P., Batory, D.: t-wise Coverage by Uniform Sampling. In: Proc. Int'l Systems and Software Product Line Conf. (SPLC), pp. 84–87. ACM, New York, NY, USA (2019)
16. Oh, J., Gazzillo, P., Batory, D., Heule, M., Myers, M.: Uniform Sampling from Kconfig Feature Models. Technical Report TR-19-02, The University of Texas at Austin, Department of Computer Science (2019)

17. Knüppel, A., Thüm, T., Mennicke, S., Meinicke, J., Schaefer, I.: Is There a Mismatch Between Real-World Feature Models and Product-Line Research? In: Proc. Europ. Software Engineering Conf./Foundations of Software Engineering (ESEC/FSE), pp. 291–302. ACM, New York, NY, USA (2017). 10.1145/3106237.3106252
18. Valiant, L.G.: The complexity of enumeration and reliability problems. SIAM J. on Computing **8**(3), 410–421 (1979)
19. Meinicke, J., Thüm, T., Schröter, R., Benduhn, F., Leich, T., Saake, G.: Mastering Software Variability with FeatureIDE. Springer, Berlin, Heidelberg, Germany (2017). https://doi.org/10.1007/978-3-319-61443-4
20. Hentze, M., Pett, T., Thüm, T., Schaefer, I.: Hyper Explanations for Feature-Model Defect Analysis. In: Proc. Int'l Working Conf. on Variability Modelling of Software-Intensive Systems (VaMoS). ACM, New York, NY, USA (2021). https://doi.org/10.1145/3442391.3442406
21. Heß, T., Sundermann, C., Thüm, T.: On the Scalability of Building Binary Decision Diagrams for Current Feature Models. In: Proc. Int'l Systems and Software Product Line Conf. (SPLC), pp. 131–135. ACM, New York, NY, USA (2021). https://doi.org/10.1145/3461001.3474452
22. Sharma, S., Gupta, R., Roy, S., Meel, K.S.: Knowledge Compilation Meets Uniform Sampling. In: Proc. Int'l Conf. on Logic for Programming, Artificial Intelligence, and Reasoning, pp. 620–636. EasyChair, Manchester, UK (2018)
23. Drechsler, R., Becker, B.: Binary Decision Diagrams: Theory and Implementation. Springer, Denver, CO, USA (1998)
24. Darwiche, A.: A Compiler for Deterministic, Decomposable Negation Normal Form. In: Proc. Conf. on Artificial Intelligence (AAAI), pp. 627–634. AAAI Press, Palo Alto, CA, USA (2002)
25. Darwiche, A., Marquis, P.: A Knowledge Compilation Map. J. Artificial Intelligence Research (JAIR) **17**(1), 229–264 (2002)
26. Sundermann, C., Nieke, M., Bittner, P.M., Heß, T., Thüm, T., Schaefer, I.: Applications of #SAT Solvers on Feature Models. In: Proc. Int'l Working Conf. on Variability Modelling of Software-Intensive Systems (VaMoS). ACM, New York, NY, USA (2021). https://doi.org/10.1145/3442391.3442404
27. Krieter, S., Thüm, T., Schulze, S., Saake, G., Leich, T.: YASA: Yet Another Sampling Algorithm. In: Proc. Int'l Working Conf. on Variability Modelling of Software-Intensive Systems (VaMoS). ACM, New York, NY, USA (2020). https://doi.org/10.1145/3377024.3377042
28. Fernández-Amorós, D., Heradio, R., Cerrada, J.A., Cerrada, C.: A Scalable Approach to Exact Model and Commonality Counting for Extended Feature Models. IEEE Trans. on Software Engineering (TSE) **40**(9), 895–910 (2014)
29. Krieter, S., Schröter, R., Thüm, T., Saake, G.: An Efficient Algorithm for Feature-Model Slicing. Technical Report FIN-001-2016, University of Magdeburg, Germany (April 2016)
30. Segura, S.: Automated Analysis of Feature Models Using Atomic Sets. In: Proc. Int'l Systems and Software Product Line Conf. (SPLC), vol. 2, pp. 201–207. IEEE, Washington, DC, USA (2008)
31. Acher, M., Heymans, P., Collet, P., Quinton, C., Lahire, P., Merle, P.: Feature Model Differences. In: Proc. Int'l Conf. on Advanced Information Systems Engineering (CAiSE), pp. 629–645. Springer, Berlin, Heidelberg, Germany (2012). https://doi.org/10.1007/978-3-642-31095-9_41
32. Johansen, M.F., Haugen, Ø., Fleurey, F.: An Algorithm for Generating T-Wise Covering Arrays from Large Feature Models. In: Proc. Int'l Systems and Software Product Line Conf. (SPLC), pp. 46–55. ACM, New York, NY, USA (2012). https://doi.org/10.1145/2362536.2362547
33. Thüm, T., Batory, D., Kästner, C.: Reasoning About Edits to Feature Models. In: Proc. Int'l Conf. on Software Engineering (ICSE), pp. 254–264. IEEE, Washington, DC, USA (2009). https://doi.org/10.1109/ICSE.2009.5070526
34. El-Sharkawy, S., Krafczyk, A., Schmid, K.: An Empirical Study of Configuration Mismatches in Linux. In: Proc. Int'l Systems and Software Product Line Conf. (SPLC), pp. 19–28. ACM, New York, NY, USA (2017). https://doi.org/10.1145/3106195.3106208
35. Heradio, R., Pérez-Morago, H.J., Fernández-Amorós, D., Bean, R., Cabrerizo, F.J., Cerrada, C., Herrera-Viedma, E.: Binary Decision Diagram Algorithms to Perform Hard Analysis Operations on Variability Models. In: Proc. Int'l Conf. on Intelligent Software Methodologies, Tools and Techniques (SOMET), pp. 139–154. IOS Press, Amsterdam, The Netherlands (2016)
36. Mendonça, M., Wąsowski, A., Czarnecki, K., Cowan, D.: Efficient Compilation Techniques for Large Scale Feature Models. In: Proc. Int'l Conf. on Generative Programming and Component Engineering (GPCE), pp. 13–22. ACM, New York, NY, USA (2008)
37. Pohl, R., Lauenroth, K., Pohl, K.: A Performance Comparison of Contemporary Algorithmic Approaches for Automated Analysis Operations on Feature Models. In: Proc. Int'l Conf. on Automated Software Engineering (ASE), pp. 313–322. IEEE, Washington, DC, USA (2011). https://doi.org/10.1109/ASE.2011.6100068

38. Baranov, E., Legay, A., Meel, K.S.: Baital: An Adaptive Weighted Sampling Approach for Improved t-Wise Coverage. In: Proc. Europ. Software Engineering Conf./Foundations of Software Engineering (ESEC/FSE), pp. 1114–1126. ACM, New York, NY, USA (2020). https://doi.org/10.1145/3368089.3409744
39. Sundermann, C., Raab, H., Heß, T., Thüm, T., Schaefer, I.: Exploiting d-DNNFs for Repetitive Counting Queries on Feature Models. Technical Report arXiv:2303.12383, Cornell University Library (March 2023). 10.48550/arXiv.2303.12383. https://arxiv.org/pdf/2303.12383
40. Bourhis, P., Duchien, L., Dusart, J., Lonca, E., Marquis, P., Quinton, C.: Reasoning on Feature Models: Compilation-Based vs. Direct Approaches. Technical Report arXiv:2302.06867, Cornell University Library (2023). https://doi.org/10.48550/arXiv.2302.06867
41. Darwiche, A.: SDD: A New Canonical Representation of Propositional Knowledge Bases, pp. 819–826. AAAI Press, Palo Alto, CA, USA (2011)
42. Sundermann, C., Heß, T., Nieke, M., Bittner, P.M., Young, J.M., Thüm, T., Schaefer, I.: Evaluating State-of-the-Art #SAT Solvers on Industrial Configuration Spaces. Empir Softw Eng (EMSE) **28** (2023). https://doi.org/10.1007/s10664-022-10265-9
43. Liang, J.H., Ganesh, V., Czarnecki, K., Raman, V.: SAT-Based Analysis of Large Real-World Feature Models Is Easy. In: Proc. Int'l Systems and Software Product Line Conf. (SPLC), pp. 91–100. Springer, Berlin, Heidelberg, Germany (2015)
44. Czarnecki, K., Wąsowski, A.: Feature Diagrams and Logics: There and Back Again. In: Proc. Int'l Systems and Software Product Line Conf. (SPLC), pp. 23–34. IEEE, Washington, DC, USA (2007)
45. Heradio, R., Fernández-Amorós, D., Mayr-Dorn, C., Egyed, A.: Supporting the Statistical Analysis of Variability Models. In: Proc. Int'l Conf. on Software Engineering (ICSE), pp. 843–853. IEEE, New York, NY, USA (2019). https://doi.org/10.1109/ICSE.2019.00091
46. Karpiński, M., Piotrów, M.: Encoding cardinality constraints using multiway merge selection networks. Constraints **24**, 234–251 (2019)
47. Moskewicz, M.W., Madigan, C.F., Zhao, Y., Zhang, L., Malik, S.: Chaff: Engineering an Efficient SAT Solver. In: Proc. Anual Conf. on Design Automation (DAC), pp. 530–535. ACM, New York, NY, USA (2001)
48. Eén, N., Sörensson, N.: An Extensible SAT-solver. In: Proc. Int'l Conf. on Theory and Applications of Satisfiability Testing (SAT), pp. 502–518. Springer, Berlin, Heidelberg, Germany (2004)
49. Sharma, S., Roy, S., Soos, M., Meel, K.S.: GANAK: A Scalable Probabilistic Exact Model Counter. In: Proc. Int'l Joint Conf. on Artificial Intelligence (IJCAI), vol. 19, pp. 1169–1176. AAAI Press, Palo Alto, CA, USA (2019)
50. Thurley, M.: sharpSAT - Counting Models with Advanced Component Caching and Implicit BCP. In: Proc. Int'l Conf. on Theory and Applications of Satisfiability Testing (SAT), pp. 424–429. Springer, Berlin, Heidelberg, Germany (2006)
51. Muise, C., McIlraith, S.A., Beck, J.C., Hsu, E.I.: Dsharp: Fast d-dnnf compilation with sharpsat. In: Kosseim, L., Inkpen, D. (eds.) Advances in Artificial Intelligence, pp. 356–361. Springer, Berlin, Heidelberg (2012)
52. Büning, H.K., Lettmann, T.: Propositional Logic: Deduction and Algorithms vol. 48. Cambridge University Press, Cambridge, United Kingdom (1999)
53. Gu, J., Purdom, P.W., Franco, J., Wah, B.W.: Algorithms for the Satisfiability (SAT) Problem: A Survey. Technical report, Cincinnati University, OH, USA (1996)
54. Valiant, L.G.: The complexity of computing the permanent. Theoretical Computer Science **8**(2), 189–201 (1979)
55. Heradio, R., Fernández-Amorós, D., Cerrada, J.A., Abad, I.: A literature review on feature diagram product counting and its usage in software product line economic models. Int. J. Softw. Eng. Knowl. Eng. (IJSEKE) **23**(08), 1177–1204 (2013)
56. Krieter, S., Thüm, T., Schulze, S., Schröter, R., Saake, G.: Propagating Configuration Decisions with Modal Implication Graphs. In: Proc. Int'l Conf. on Software Engineering (ICSE), pp. 898–909. ACM, New York, NY, USA (2018). https://doi.org/10.1145/3180155.3180159
57. Oh, J., Gazzillo, P., Batory, D., Heule, M., Myers, M.: Scalable Uniform Sampling for Real-World Software Product Lines. Technical Report TR-20-01, The University of Texas at Austin, Department of Computer Science (2020)
58. Thüm, T., Kästner, C., Benduhn, F., Meinicke, J., Saake, G., Leich, T.: FeatureIDE: An extensible framework for feature-oriented software development. Science of Computer Programming (SCP) **79**, 70–85 (2014). https://doi.org/10.1016/j.scico.2012.06.002
59. Zhang, W., Zhao, H., Mei, H.: A Propositional Logic-Based Method for Verification of Feature Models. In: Proc. Int'l Conf. on Formal Engineering Methods (ICFEM), pp. 115–130. Springer, Berlin, Heidelberg, Germany (2004)

60. Halin, A., Nuttinck, A., Acher, M., Devroey, X., Perrouin, G., Baudry, B.: Test them all, is it worth it? Assessing configuration sampling on the JHipster web development stack. Empir. Softw. Eng. (EMSE) **24**(2), 674–717 (2019)
61. Janota, M., Lynce, I., Marques-Silva, J.: Algorithms for computing backbones of propositional formulae. AI Commun. **28**(2), 161–177 (2015)
62. Al-Hajjaji, M., Krieter, S., Thüm, T., Lochau, M., Saake, G.: IncLing: Efficient Product-line Testing Using Incremental Pairwise Sampling. In: Proc. Int'l Conf. on Generative Programming: Concepts & Experiences (GPCE), pp. 144–155. ACM, New York, NY, USA (2016). https://doi.org/10.1145/2993236.2993253
63. Haslinger, E.N., Lopez-Herrejon, R.E., Egyed, A.: Using Feature Model Knowledge to Speed Up the Generation of Covering Arrays. In: Proc. Int'l Workshop on Variability Modelling of Software-Intensive Systems (VaMoS), pp. 16–1166. ACM, New York, NY, USA (2013). https://doi.org/10.1145/2430502.2430524
64. Johansen, M.F., Haugen, Ø., Fleurey, F.: Properties of Realistic Feature Models Make Combinatorial Testing of Product Lines Feasible. In: Proc. Int'l Conf. on Model Driven Engineering Languages and Systems (MODELS), pp. 638–652. Springer, Berlin, Heidelberg, Germany (2011). https://doi.org/10.1007/978-3-642-24485-8_47
65. Johansen, M.F., Haugen, Ø., Fleurey, F., Eldegard, A.G., Syversen, T.: Generating Better Partial Covering Arrays by Modeling Weights on Sub-Product Lines. In: Proc. Int'l Conf. on Model Driven Engineering Languages and Systems (MODELS), pp. 269–284. Springer, Berlin, Heidelberg, Germany (2012). https://doi.org/10.1007/978-3-642-33666-9_18
66. Kowal, M., Schulze, S., Schaefer, I.: Towards Efficient SPL Testing by Variant Reduction. In: Proc. Int'l Workshop on Variability and Composition (VariComp), pp. 1–6. ACM, New York, NY, USA (2013). https://doi.org/10.1145/2451617.2451619
67. Oster, S., Markert, F., Ritter, P.: Automated Incremental Pairwise Testing of Software Product Lines. In: Proc. Int'l Systems and Software Product Line Conf. (SPLC), pp. 196–210. Springer, Berlin, Heidelberg, Germany (2010). https://doi.org/10.1007/978-3-642-15579-6_14
68. Czarnecki, K., Kim, C.H.P.: Cardinality-Based Feature Modeling and Constraints: A Progress Report. In: Proc. Int'l Workshop on Software Factories (SF), pp. 16–20 (2005)
69. Bagheri, E., Noia, T.D., Gasevic, D., Ragone, A.: Formalizing interactive staged feature model configuration. J. Software: Evolution and Process **24**(4), 375–400 (2012)
70. Clements, P.C., McGregor, J.D., Cohen, S.G.: The Structured Intuitive Model for Product Line Economics (SIMPLE). Technical report, Carnegie-Mellon University, PA, USA (2005)
71. Cohen, S.: Predicting When Product Line Investment Pays. Technical report, Carnegie-Mellon University, PY, USA (2003)
72. Chen, S., Erwig, M.: Optimizing the Product Derivation Process. In: Proc. Int'l Systems and Software Product Line Conf. (SPLC), pp. 35–44. IEEE, Washington, DC, USA (2011). https://doi.org/10.1109/SPLC.2011.47
73. Mazo, R., Dumitrescu, C., Salinesi, C., Diaz, D.: Recommendation Heuristics for Improving Product Line Configuration Processes. In: Recommendation Systems in Software Engineering, pp. 511–537. Springer, Berlin, Heidelberg, Germany (2014)
74. Oh, J., Batory, D., Myers, M., Siegmund, N.: Finding Near-Optimal Configurations in Product Lines by Random Sampling. In: Proc. Int'l Symposium on Foundations of Software Engineering (FSE), pp. 61–71 (2017). https://doi.org/10.1145/3106237.3106273
75. Aziz, R.A., Chu, G., Muise, C., Stuckey, P.: #∃ SAT: Projected Model Counting. In: Proc. Int'l Conf. on Theory and Applications of Satisfiability Testing (SAT), pp. 121–137. Springer, Berlin, Heidelberg, Germany (2015)
76. Acher, M., Collet, P., Lahire, P., France, R.B.: Slicing Feature Models. In: Proc. Int'l Conf. on Automated Software Engineering (ASE), pp. 424–427. IEEE, Washington, DC, USA (2011). https://doi.org/10.1109/ASE.2011.6100089
77. Krieter, S., Schröter, R., Thüm, T., Fenske, W., Saake, G.: Comparing Algorithms for Efficient Feature-Model Slicing. In: Proc. Int'l Systems and Software Product Line Conf. (SPLC), pp. 60–64. ACM, New York, NY, USA (2016). https://doi.org/10.1145/2934466.2934477
78. Fichte, J.K., Hecher, M., Morak, M., Woltran, S.: Exploiting Treewidth for Projected Model Counting and Its Limits. In: Proc. Int'l Conf. on Theory and Applications of Satisfiability Testing (SAT), pp. 165–184. Springer, Cham, Switzerland (2018)
79. Acher, M., Collet, P., Lahire, P., France, R.B.: Comparing Approaches to Implement Feature Model Composition. In: Proc. Europ. Conf. on Modelling Foundations and Applications (ECMFA), pp. 3–19. Springer, Berlin, Heidelberg, Germany (2010)

80. Rice, M., Kulhari, S.: A Survey of Static Variable Ordering Heuristics for Efficient BDD/MDD Construction. Technical report, University of California, Riverside, CA, USA (2008)
81. Dowling, W.F., Gallier, J.H.: Linear-time algorithms for testing the satisfiability of propositional horn formulae. J. Log. Program. **1**(3), 267–284 (1984). https://doi.org/10.1016/0743-1066(84)90014-1
82. Wang, J., Gu, W., Yin, M., Wang, D.: MCN and MO: Two Heuristic Strategies in Knowledge Compilation Using Extension Rule. In: International Conference on Signal Processing Systems (ICSPS), pp. 389–393 (2009). https://doi.org/10.1109/ICSPS.2009.99
83. Huth, M., Ryan, M.: Logic in Computer Science: Modelling and Reasoning About Systems. Cambridge University Press, Cambridge, United Kingdom (2004)
84. Pipatsrisawat, K., Darwiche, A.: New Compilation Languages Based on Structured Decomposability. In: Proc. Conf. on Artificial Intelligence (AAAI), vol. 8, pp. 517–522. AAAI Press, Palo Alto, CA, USA (2008)
85. Darwiche, A.: Decomposable negation normal form. J. ACM **48**(4), 608–647 (2001)
86. Darwiche, A.: New Advances in Compiling CNF to Decomposable Negation Normal Form. In: Proc. Europ. Conf. on Artificial Intelligence, pp. 318–322. IOS Press, Amsterdam, The Netherlands (2004)
87. Lagniez, J.-M., Marquis, P.: An Improved Decision-DNNF Compiler. In: Proc. Int'l Joint Conf. on Artificial Intelligence (IJCAI), pp. 667–673. AAAI Press, Palo Alto, CA, USA (2017)
88. Muise, C., McIlraith, S., Beck, J.C., Hsu, E.: Fast d-DNNF Compilation with sharpSAT. In: Proc. Conf. on Artificial Intelligence (AAAI). AAAI Press, Palo Alto, CA, USA (2010)
89. Koriche, F., Lagniez, J.-M., Marquis, P., Thomas, S.: Knowledge Compilation for Model Counting: Affine Decision Trees. In: Proc. Int'l Joint Conf. on Artificial Intelligence (IJCAI). AAAI Press, Palo Alto, CA, USA (2013)
90. Van den Broeck, G., Darwiche, A.: On the role of canonicity in knowledge compilation. Proc. Conf. on Artificial Intelligence (AAAI) **29**(1) (2015). https://doi.org/10.1609/aaai.v29i1.9423
91. Nishino, M., Yasuda, N., Minato, S.-i., Nagata, M.: Zero-Suppressed Sentential Decision Diagrams. In: Proc. Conf. on Artificial Intelligence (AAAI), pp. 1058–1066. AAAI Press, Palo Alto, CA, USA (2016)
92. Oztok, U., Darwiche, A.: An exhaustive DPLL algorithm for model counting **62**(1), 1–32 (2018). https://doi.org/10.1613/jair.1.11201
93. Bryant, R.E.: Graph-based algorithms for boolean function manipulation. IEEE Trans. on Computers **C-35**(8), 677–691 (1986). https://doi.org/10.1109/tc.1986.1676819
94. Jha, A., Suciu, D.: Knowledge compilation meets database theory: Compiling queries to decision diagrams. In: Proceedings of the 14th International Conference on Database Theory (ICDT), pp. 162–173. ACM, New York, NY, USA (2011). https://doi.org/10.1145/1938551.1938574
95. Minato, S.-I.: Zero-Suppressed BDDs for Set Manipulation in Combinatorial Problems. In: Proc. Anual Conf. on Design Automation (DAC), pp. 272–277. ACM, New York, NY, USA (1993)
96. Razgon, I.: On oblivious branching programs with bounded repetition that cannot efficiently compute CNFS of bounded treewidth. Theory of Computing Systems **61**(3), 755–776 (2017)
97. Vinkhuijzen, L., Laarman, A.: The Power of Disjoint Support Decompositions in Decision Diagrams. In: NASA Formal Methods Symposium (NFM), pp. 790–799. Springer, Berlin, Heidelberg, Germany (2022)
98. van Dijk, T., Wille, R., Meolic, R.: Tagged BDDs: Combining Reduction Rules from Different Decision Diagram Types. In: Proc. Int'l Conf. on Formal Methods in Computer-Aided Design (FMCAD), Washington, DC, USA, pp. 108–115 (2017). IEEE
99. Bova, S.: Sdds are exponentially more succinct than obdds. Proc. Conf. on Artificial Intelligence (AAAI) 30(1) (2016). https://doi.org/10.1609/aaai.v30i1.10107
100. Oztok, U., Darwiche, A.: On Compiling CNF into Decision-DNNF. In: Proc. Int'l Conf. on Principles and Practice of Constraint Programming (CP), pp. 42–57. Springer, Berlin, Heidelberg, Germany (2014)
101. Pett, T., Krieter, S., Thüm, T., Lochau, M., Schaefer, I.: AutoSMP: An Evaluation Platform for Sampling Algorithms. In: Proc. Int'l Systems and Software Product Line Conf. (SPLC), pp. 41–44. ACM, New York, NY, USA (2021). https://doi.org/10.1145/3461002.3473073
102. Nieke, M., Mauro, J., Seidl, C., Thüm, T., Yu, I.C., Franzke, F.: Anomaly Analyses for Feature-Model Evolution. In: Proc. Int'l Conf. on Generative Programming: Concepts & Experiences (GPCE), pp. 188–201. ACM, New York, NY, USA (2018). https://doi.org/10.1145/3278122.3278123
103. Pett, T., Thüm, T., Runge, T., Krieter, S., Lochau, M., Schaefer, I.: Product Sampling for Product Lines: The Scalability Challenge. In: Proc. Int'l Systems and Software Product Line Conf. (SPLC), pp. 78–83. ACM, New York, NY, USA (2019). https://doi.org/10.1145/3336294.3336322
104. Oztok, U., Darwiche, A.: A Top-Down Compiler for Sentential Decision Diagrams. In: Proc. Int'l Joint Conf. on Artificial Intelligence (IJCAI), pp. 3141–3148. AAAI Press, Palo Alto, CA, USA (2015)
105. Somenzi, F.: Efficient Manipulation of Decision Diagrams. Int. J. Softw. Tools Technol. Trans. (STTT) **3**(2), 171–181 (2001). https://doi.org/10.1007/s100090100042

106. Janssen, G.: A Consumer Report on BDD Packages. In: Proc. Symposium on Integrated Circuits and Systems Design (SBCCI), pp. 217–222. IEEE, Los Alamitos, CA, USA (2003)
107. Toda, T., Soh, T.: Implementing Efficient All Solutions SAT Solvers. ACM J. of Experimental Algorithmics (JEA) **21**(1), 1–12111244 (2016). https://doi.org/10.1145/2975585
108. Pett, T., Krieter, S., Runge, T., Thüm, T., Lochau, M., Schaefer, I.: Stability of Product-Line Sampling in Continuous Integration. In: Proc. Int'l Working Conf. on Variability Modelling of Software-Intensive Systems (VaMoS). ACM, New York, NY, USA (2021). https://doi.org/10.1145/3442391.3442410
109. Hutter, F., Hoos, H.H., Leyton-Brown, K.: Sequential model-based optimization for general algorithm configuration. In: Learning and Intelligent Optimization: 5th International Conference, LION 5, Rome, Italy, January 17-21, 2011. Selected Papers 5, pp. 507–523 (2011). Springer
110. Coy, S.P., Golden, B.L., Runger, G.C., Wasil, E.A.: Using Experimental Design to Find Effective Parameter Settings for Heuristics. Journal of Heuristics **7**, 77–97 (2001)
111. Tseytin, G.S.: On the Complexity of Derivation in Propositional Calculus, pp. 466–483. Springer, Berlin, Heidelberg, Germany (1983). https://doi.org/10.1007/978-3-642-81955-1_28
112. Voronov, A., Åkesson, K., Ekstedt, F.: Enumeration of Valid Partial Configurations. In: Proc. Configuration Workshop (ConfWS), vol. 755, pp. 25–31. ceur-ws.org, Aachen, Germany (2011)
113. Galindo, J.A., Benavides, D., Trinidad, P., Gutiírrez-Fernández, A.-M., Ruiz-Cortés, A.: Automated analysis of feature models: Quo Vadis? Computing **101**(5), 387–433 (2019). https://doi.org/10.1007/s00607-018-0646-1
114. Janota, M., Kiniry, J., Botterweck, G.: Formal Methods in Software Product Lines: Concepts, Survey, and Guidelines. Technical Report Lero-TR-SPL-2008-02, Lero, University of Limerick (May 2008)
115. Lutz, R.: Survey of Product-Line Verification and Validation Techniques. Technical Report 2014/41221, NASA, Jet Propulsion Laboratory, La Canada Flintridge, CA, USA (May 2007)
116. Schobbens, P.-Y., Heymans, P., Trigaux, J.-C.: Feature Diagrams: A Survey and a Formal Semantics. In: Proc. Int'l Conf. on Requirements Engineering (RE), pp. 136–145. IEEE, Washington, DC, USA (2006). https://doi.org/10.1109/RE.2006.23
117. Varshosaz, M., Al-Hajjaji, M., Thüm, T., Runge, T., Mousavi, M.R., Schaefer, I.: A Classification of Product Sampling for Software Product Lines. In: Proc. Int'l Systems and Software Product Line Conf. (SPLC), pp. 1–13. ACM, New York, NY, USA (2018). https://doi.org/10.1145/3233027.3233035
118. Medeiros, F., Kästner, C., Ribeiro, M., Gheyi, R., Apel, S.: A Comparison of 10 Sampling Algorithms for Configurable Systems. In: Proc. Int'l Conf. on Software Engineering (ICSE), pp. 643–654. ACM, New York, NY, USA (2016). https://doi.org/10.1145/2884781.2884793

Publisher's Note Springer Nature remains neutral with regard to jurisdictional claims in published maps and institutional affiliations.

C.3 EMSE'23: Evaluating State-of-the-Art #SAT Solvers on Industrial Configuration Spaces

[B1] Chico Sundermann, Tobias Heß, Michael Nieke, Paul Maximilian Bittner, Jeffrey M. Young, Thomas Thüm, and Ina Schaefer. "Evaluating State-of-the-Art #SAT Solvers on Industrial Configuration Spaces". In: *Empirical Software Engineering (EMSE)* 28.2 (2023), p. 38. DOI: 10.1007/s10664-022-10265-9

Published under Creative Commons Attribution 4.0 International (CC-BY), see https://creativecommons.org/licenses/by/4.0/.

Empirical Software Engineering (2023) 28:29
https://doi.org/10.1007/s10664-022-10265-9

Evaluating state-of-the-art #SAT solvers on industrial configuration spaces

Chico Sundermann[1] · Tobias Heß[1] · Michael Nieke[2] · Paul Maximilian Bittner[1] · Jeffrey M. Young[3] · Thomas Thüm[1] · Ina Schaefer[2]

Accepted: 25 November 2022 / Published online: 13 January 2023
© The Author(s) 2023

Abstract

Product lines are widely used to manage families of products that share a common base of features. Typically, not every combination (configuration) of features is valid. Feature models are a de facto standard to specify valid configurations and allow standardized analyses on the variability of the underlying system. A large variety of such analyses depends on computing the number of valid configurations. To analyze feature models, they are typically translated to propositional logic. This allows to employ #SAT solvers that compute the number of satisfying assignments of the propositional formula translated from a feature model. However, the #SAT problem is generally assumed to be even harder than SAT and its scalability when applied to feature models has only been explored sparsely. Our main contribution is an investigation of the performance of off-the-shelf #SAT solvers on computing the number of valid configurations for industrial feature models. We empirically evaluate 21 publicly available #SAT solvers on 130 feature models from 15 subject systems. Our results indicate that current solvers master a majority of the evaluated systems (13/15) with the fastest solvers requiring less than one second for each successfully evaluated feature model. However, there are two complex systems for which none of the evaluated solvers scales. For the given experiment design, the solvers that consumed the least runtime are `sharpSAT` (2.5 seconds in sum for the 13 systems) and `Ganak` (3.5 seconds).

Keywords Configurable systems · Feature models · Product lines · Model counting · Configuration counting · #SAT · Benchmark

1 Introduction

A product line represents a family of products that share certain configuration options, also called features (Benavides et al. 2010; Sobernig et al. 2016; Bosch et al. 2001; Heradio et al.

Communicated by: Philippe Collet, Sarah Nadi, Christoph Seidl

This article belongs to the Topical Collection: *Special Issue on Software Product Lines and Variability-rich Systems (SPLC).*

✉ Chico Sundermann
chico.sundermann@uni-ulm.de

Extended author information available on the last page of the article.

2019). Each product is composed of a distinct selection of features, called configuration (Apel et al. 2013). However, systems typically contain constraints which limit the set of valid configurations (e.g., the selection of one feature requires selecting another feature). These constraints are typically specified as a feature model (Bagheri et al. 2012; Batory 2005; Czarnecki and Wąsowski 2007) which consists of a tree hierarchy and additional cross-tree constraints.

Managing a product line is typically complex due to the high number of constraints (Benavides et al. 2010). For example, one feature model we analyzed, representing an automotive product line, contains more than 10,000 cross-tree constraints in addition to hierarchical constraints. Manually keeping track of all these dependencies is infeasible (Sprey et al. 2020). Consequently, a large variety of automated support in terms of analyses has been proposed (Batory 2005; Schröter et al. 2016; Mendonça et al. 2009; Pohl et al. 2011; Czarnecki and Wąsowski 2007; Perrouin et al. 2010; Segura 2008; Galindo et al. 2016; Benavides et al. 2010; Sprey et al. 2020). A multitude of analyses is based on feature-model counting (i.e., computing the number of valid configurations), such as uniform random sampling (Munoz et al. 2019; Oh et al. 2019; Sharma et al. 2018) and detecting design errors (Sundermann et al. 2021; Heradio et al. 2013; Chen and Erwig 2011; Fernández-Amorós et al. 2014; Heradio-Gil et al. 2011; Kübler et al. 2010). We refer to the number of valid configurations of a feature model as its *cardinality* (Sundermann et al. 2021).

In the literature, the scalability of analyses that depend on computing the number of valid configurations is largely unknown. Existing work either focuses on single analyses (e.g., uniform random sampling of feature-model configurations (Oh et al. 2017, 2019; Sharma et al. 2018)), has not been evaluated on industrial feature models (Heradio-Gil et al. 2011; Fernández-Amorós et al. 2014; Pohl et al. 2011), or considers very few solvers or systems (Kübler et al. 2010; Oh et al. 2017, 2019; Sharma et al. 2018). In this paper, we focus on propositional model counting (for short #SAT) which determines the number of satisfying assignments for a given propositional formula. As the translation of feature models to propositional logic is well-researched (Benavides et al. 2010; Batory 2005), #SAT solvers can be applied out of the box to compute the cardinality of feature models. However, #SAT is at least as hard as SAT because after computing #SAT (i.e., the number of satisfying assignments) it is trivial to determine whether a formula is SAT (i.e., there is at least one satisfying assignment). In general, #SAT is assumed to be harder (Burchard et al. 2015; Valiant 1979). While it is widely accepted that regular SAT is typically easy for industrial feature models (compared to randomly generated formulas (Pett et al. 2019; Mendonça et al. 2009)), this has not been explored for #SAT.

In this work, we provide insights on the scalability of modern off-the-shelf #SAT solvers for the analysis of feature models. Analyses based on feature-model counting can only be applied in practice if available #SAT solvers scale to industrial feature models considering time restrictions for typical use cases, such as interactive settings (Fritsch et al. 2020; Sprey et al. 2020; Krieter et al. 2017; Acher et al. 2013; Benavides et al. 2007) or continuous integration environments (Pett et al. 2021). We thus evaluate the runtimes of analyzing feature models with publicly available #SAT solvers. Furthermore, we provide recommendations on which solvers to use for analyzing feature models to reduce runtimes.

#SAT solvers rely on a variety of techniques to compute the number of satisfying assignments. While some solvers only report the number of satisfying assignments (Bayardo Jr and Pehoushek 2000; Sang et al. 2004; Thurley 2006; Burchard et al. 2015; Biere 2008), other solvers apply knowledge compilation to different target languages, such as *binary decision diagrams* (BDDs) http://buddy.sourceforge.net/manual/main.html. Accessed: 02

Mar 2020; https://github.com/vscosta/cudd. Accessed: 13 Jun 2020 (Toda and Soh 2016), *deterministic decomposable negation normal forms* (d-DNNFs) (Darwiche 2004; Lagniez and Marquis 2017; Muise et al. 2010), *sentential decision diagrams* (SDDs) (Oztok and Darwiche 2015), and *extended affine decision trees* (Koriche et al. 2013). The compiled target languages may be reused for further feature-model analyses. We analyze the benefits of different techniques to identify promising classes of #SAT solvers.

In general, the runtime required to analyze a feature model depends on structural properties related to its size and complexity (Mendonça et al. 2009; Kübler et al. 2010; Fernández-Amorós et al. 2014). We provide first insights on properties which induce a time-consuming computation for every or some #SAT solvers. In particular, we analyze the correlation between the runtimes and a variety of structural metrics.

For some feature models, it may be infeasible to compute an exact result using publicly available solvers. In this case, approximate #SAT solvers, which estimate the number of satisfying assignments for a given formula, may be beneficial. We inspect the benefits of approximate #SAT solvers when applied to industrial feature models.

Overall, we evaluate 19 exact and 2 approximate off-the-shelf #SAT solvers which are publicly available. For our empirical evaluation, we consider 15 subject systems with overall 130 feature models. We provide the framework and data used for the empirical evaluation on Zenodo.[1] In particular, our work provides the following contributions:

1. We examine the performance regarding runtime of #SAT technology on 130 industrial feature models.
2. We identify best performing #SAT solvers out of 21 off-the-shelf tools.
3. We compare the benefits of different #SAT technologies.
4. We examine the correlation between the runtime of #SAT solvers and structural metrics of the feature model.
5. We inspect the performance of two approximate #SAT solvers.
6. We provide the number of valid configurations for feature models in our dataset.

In this work, we extend our previous conference publication (Sundermann et al. 2020) regarding the following aspects. First, we additionally evaluate ten more exact #SAT solvers. Second, we examine the runtimes of two approximate #SAT solvers. Third, we consider four additional subject systems. Fourth, we analyze the correlation between the runtime and 12 structural metrics of the feature models. Fifth, we improve the accuracy of our results by repeating the measurements and applying statistical tests to study the significance of our results. Overall, the evaluation subsumes the previous evaluation (Sundermann et al. 2020) except for analyzing the evolution of systems. We consider a more thorough analysis (compared to the previous evaluation (Sundermann et al. 2020)) of the evolution as out of scope for this work.

2 Motivating example

Figure 1 shows a feature diagram representing a simplified car product line. A feature diagram is a commonly used visual representation of a feature model (Benavides et al. 2010). It visualizes the feature model's tree structure and additional cross-tree constraints given in

[1] https://doi.org/10.5281/zenodo.7329979

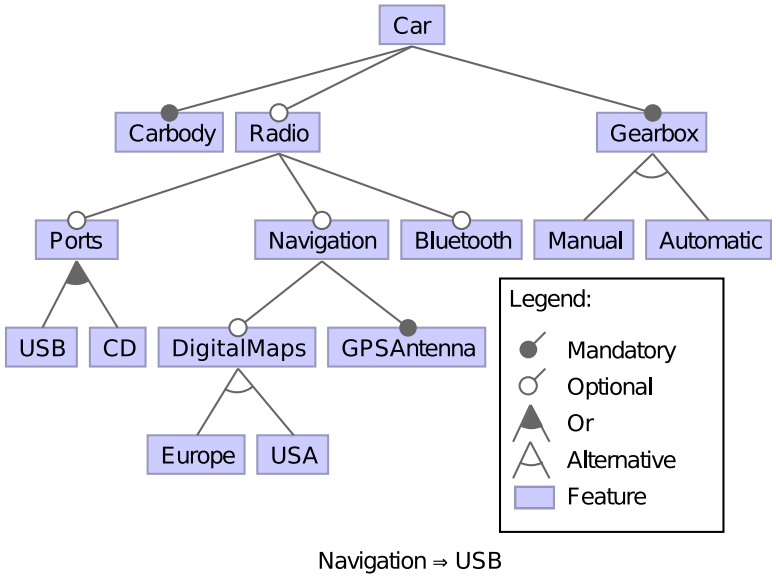


Fig. 1 Example feature model adapted from Ananieva et al. (Ananieva et al. 2016)

propositional logic. The tree structure and the cross-tree constraints specify the set of valid configurations.

In our example, each car of the product line requires a *Carbody*. This is indicated by the mandatory property of the feature. In contrast, a *Radio* is an optional feature (i.e., it may or may not be selected). A configuration that does not contain exactly one of the *Gearbox* types, *Manual* or *Automatic*, is invalid, as they appear in an alternative-relation in the feature diagram. Furthermore, the *Ports* of a *Radio* include at least one of *USB* or *CD*. This relation is described by an or-relation. The cross-tree constraint $Navigation \Rightarrow USB$ represents that a car with *Navigation* requires a *USB* port.

To analyze a feature model, we can use its cardinality (i.e., the number of valid configurations). Consider the following scenario. The vast majority of automatic cars are sold in the USA. As a consequence, a developer introduces a new constraint $Automatic \Rightarrow USA$ (automatic cars require digital maps for the USA). Using a #SAT solver, the developer finds that the cardinality is 42 before the change and 25 afterwards. The immense decrease in the cardinality, if unexpected, may already be an indicator for a design problem, because the set of available cars is almost halved. Further, the cardinality can be combined with domain knowledge for more sophisticated insights as follows. In the old version, there are 21 cars with an *Automatic* gearbox and 21 cars with a *Manual* gearbox. The newly introduced constraint has no impact on cars with *Manual*. Thus, there are still 21 cars with *Manual* in the new version which implies that only $25 - 21 = 4$ valid configurations with *Automatic* remain. Due to the tree hierarchy, the introduced constraint ($Automatic \Rightarrow USA$) requires each automatic car to also have *Radio*, *Navigation*, *DigitalMaps*, and *USB*. This side effect was probably unintended and can be fixed by changing the constraint to $Automatic \wedge DigitalMaps \Rightarrow USA$. While the original constraint ($Automatic \Rightarrow USA$) induces an immense and possibly unintended reduction in the variability, it introduces no traditional anomalies (e.g., core, false-optional, or dead features; cf. (Benavides et al. 2010) for more details). Hence, in such cases it is hard to detect the side effects with traditional SAT-based analyses. In the provided scenario, we used the variability reduction of a feature model update to detect side effects, one of 21 applications of #SAT we identified in previous work (Sundermann et al. 2021).

Without cross-tree constraints, computing the number of valid configurations has linear time complexity in the number of features (Heradio-Gil et al. 2011). Only considering the tree-structure, the selections in a subtree are completely independent of selections in other subtrees. Therefore, the cardinality of each subtree can be computed separately. The cardinality of the feature model can be computed by traversing the tree once and applying rules for each relation type, recursively. For example, the cardinality of an alternative group is equal to the sum of cardinalities of the subtrees induced by the children of the alternative group. However, for feature models with cross-tree constraints, the proposed procedure results in a wrong cardinality as interdependencies are disregarded. Hence, a more sophisticated algorithm is required.

With cross-tree constraints, the number of configurations cannot be computed in linear time complexity w.r.t. to the number of features. Every feature model can be translated to a propositional formula (Mendonça et al. 2009). Furthermore, a feature model that contains cross-tree constraints can represent every propositional formula and vice versa (Knüppel et al. 2017). Thus, computing the satisfiability of a model with those constraints is as hard as SAT and computing the number of valid configurations is as hard as #SAT.

3 The need for feature-model counting

In our previous work (Sundermann et al. 2021), we surveyed a large variety of applications dependent on the number of valid configurations of feature models. The presented applications indicate the benefits of applying #SAT to feature models for multiple aspects, such as detecting design errors, economical estimations, and guidance for developers. Overall, we found 21 applications gathered from the literature or inspired by industry projects, one of which we exemplified in the last section. In the following, we present some exemplary applications that depend on computing the number of valid configurations provided in the original work (Sundermann et al. 2021). Each of the exemplary applications is inspired by insights of our industry projects.

Variability reduction In Section 2, we already introduced an example of variability reduction to detect the side effect of a new constraint. Generally, when working with product lines, it is infeasible to manually keep track of all possible side effects when applying changes (Sprey et al. 2020; Chen and Erwig 2011; Heradio et al. 2016). These side effects are especially difficult to detect if they introduce no traditional anomaly, such as dead features (Benavides et al. 2010), or a void feature model (Benavides et al. 2010) (i.e., the feature model does not describe a single valid configuration). In such cases, computing the cardinality before and after a change may provide an indicator for faulty edits (Sundermann et al. 2021). Another use case is willingly decreasing the cardinality to limit the variability of a system during an evolution. However, in order to grasp the impact of such changes it is necessary to know the cardinality before and after the change (Benavides et al. 2005).

Feature prioritization In some scenarios, features can be prioritized based on the number of valid configurations they appear in. For example, a developer may have to decide which feature to develop next. Suppose the developer's goal is to develop as many distinct products as possible. Consequently, the developer wants to prioritize features that appear in a higher number of valid configurations, which can be computed using a #SAT solver (Sundermann et al. 2021).

Uniform random sampling As it is mostly infeasible to analyze a configuration space by considering each single configuration, it is common to create representative samples for a product line (Munoz et al. 2019). However, finding these samples is not trivial (Oh et al. 2016, 2017). Uniform random sampling creates representative (i.e., each valid configuration has the same chance to be included) samples (Munoz et al. 2019). One technique for uniform random sampling is to create a bijection between integers and valid configurations. Suppose the cardinality of the feature model is #FM. Then, by randomly selecting an integer within the range [1,. . .,#FM], each configuration has the same probability to be included in the sample. The bijection can be achieved using #SAT by recursively assigning the features (Munoz et al. 2019). Following the algorithm of Munoz et al. (Oh et al. 2019), the number of valid configurations needs to be computed for each assignment in the worst case. This requires an efficient #SAT solver, especially for large systems (Munoz et al. 2019).

4 Propositional model counting

In this section, we provide some background for propositional logic, the #SAT problem, and different strategies employed by the evaluated solvers. Note that this section is not necessary to understand the empirical evaluation. Hence, the section can be skipped if considering the evaluated solvers as black boxes is sufficient for the reader.

Let F be a propositional formula and $vars(F)$ the corresponding set of variables with $|vars(F)| = n$. An assignment is a function $\alpha : vars(F) \rightarrow \{0, 1, undef\}$ that maps variables contained in F to the truth values (0 or 1) or undefined ($undef$) (Kübler et al. 2010). Assignments can be partial, meaning that some variables $v \in vars(F)$ are mapped to $undef$. Otherwise, the assignment is called full (Kübler et al. 2010). For an assignment α, $|\alpha| \leq n$ corresponds to the number of variables mapped to 0 or 1 in α. We use $F(\alpha) \in \{0, 1\}$ to denote whether a full assignment α satisfies the formula F. We refer to assignments α with $F(\alpha) = 1$ as satisfying.

Propositional model counting (for short #SAT) is defined as the problem of computing the number of satisfying full assignments of a propositional formula (Gomes et al. 2006; Kübler et al. 2010). $\#F = |\{\alpha \mid F(\alpha) = 1\}|$ corresponds to the number of satisfying full assignments of formula F. In the following, we present three popular model counting methods employed by the majority of solvers in our empirical evaluation, namely (Davis-Putnam-Logemann-Loveland) DPLL-based (Bayardo Jr and Pehoushek 2000; Sang et al. 2005; Thurley 2006; Burchard et al. 2015; Biere 2008), d-DNNF-based (Darwiche 2004; Lagniez and Marquis 2017; Muise et al. 2010), and BDD-based (Toda and Soh 2016) counting.

The algorithms based on exhaustive DPLL iteratively assign variables to ultimately compute the number of satisfying assignments. The goal is to find an assignment that either satisfies or does not satisfy the formula for each possible assignment of the remaining $n - |\alpha|$ variables. If the formula evaluates to false under α, the number of resulting satisfying assignments for α is 0. If it evaluates to true, the number of satisfying assignments for α is $2^{n-|\alpha|}$, which is the number of possible assignments of the remaining variables. In particular, a satisfying full assignment induces exactly $2^{n-n} = 1$ solution. After computing a result for α, DPLL uses backtracking to find remaining assignments. The backtracking algorithm is performed until each satisfying assignment is covered. The sum of computed results is the exact number of satisfying assignments (Biere et al. 2009).

Another possible way to compute the number of satisfying assignments are d-DNNFs. The term d-DNNF stands for deterministic, decomposable negation normal form (Darwiche and Marquis 2002). A formula is in *negation normal form* (NNF) if the logical operators are limited to $\wedge$ (conjunction), $\vee$ (disjunctions), $\neg$ (negations) and negations only appear directly in front of literals (Huth and Ryan 2004). A formula F is called *deterministic* if each child $D_1, \ldots, D_n$ of a disjunction $D \in F$ is logically disjunct (i.e., $\forall i, j : i \neq j : D_i \wedge D_j \models \bot$) (Darwiche and Marquis 2002). Determinism implies that the children $D_1, \ldots, D_n$ of a disjunction D share no common solutions. Therefore, the number of satisfying assignments of the disjunction is equal to the sum of its children's results (i.e., $\#D = \sum_{i=1}^{n} \#D_i$) (Biere et al. 2009). A formula is called *decomposable* if the children $C_1, \ldots, C_n$ of a conjunction C share no variables (i.e., $\forall i, j : i \neq j : vars(C_i) \cap vars(C_j) = \emptyset$) (Darwiche and Marquis 2002). Decomposability implies that assignments for variables of the children $C_1, \ldots, C_n$ are independent of each other as the variables are disjoint. It follows that the number of satisfying assignments of the conjunction is equal to the product of the results for each child (i.e., $\#C = \prod_{i=1}^{n} \#C_i$) (Biere et al. 2009). Using both properties (determinism and decomposability), it is possible to compute the overall number of satisfying assignments by traversing the formula once (Biere et al. 2009). d-DNNF-based #SAT solving corresponds to compiling a propositional formula to d-DNNF and then retrieving the number of satisfying assignments by traversing the d-DNNF. After the compilation, computing the model count takes linear time w.r.t. the number of the d-DNNF nodes (Darwiche 2004).

Finally, #SAT may also be computed using a binary decision diagram $BDD(F)$ representing the propositional formula F. A binary decision diagram is a rooted directed acyclic graph two terminal nodes $\bot$ and $\top$. Every non-terminal node x is associated with a variable $v \in vars(F)$ and has precisely two outgoing edges, named *low* (setting v to false) and *high* (setting v to true). Typically, one considers reduced ordered binary decision diagrams (BDD) (Bryant 1986, 2018; Mendonça 2009). A BDD is *ordered*, if nodes associated to a variable v_i always precede nodes associated to a variable v_j or vice versa. If a BDD is *reduced*, then it (1) does not contain nodes with their low and high edges incident with the same node and (2) no two nodes associated with the same variable have the same nodes incident to their low and high edges. All satisfying assignments for F correspond to a path P from the root node to $\top$ (1-path) in $BDD(F)$. Let x_v be a node associated to the variable v. If the low edge of x_v is contained in P, then v is set to false. Analogously, v is set to true if P contains the high edge of x_v. If no node associated to v is contained in P, then v can be assigned an arbitrary value. Consequently, every 1-path induces $2^{n-|P|}$ satisfying assignments, where $|P|$ is the number of edges in P, and it suffices to iterate over all 1-paths in *BDD(F)* to compute $\#F$:

$$\#F = \sum_{\substack{\text{1-path } P \\ \text{in } BDD(F)}} 2^{n-|P|} \tag{1}$$

This can be achieved in linear time with respect to the number of nodes in $BDD(F)$ (Bryant 2018). BDDs are known to be sensitive to the order of variables and there are examples in which one order results in a BDD with linear number of nodes (w.r.t. the number of features) and another order results in a BDD of exponential size (Bryant 1986).

The main difference between the solving techniques is the reuse of results. DPLL-based solvers perform a single computation and typically just return the number of satisfying assignments (Bayardo Jr and Pehoushek 2000; Sang et al. 2004; Thurley 2006; Burchard et al. 2015; Biere 2008). When using a BDD or d-DNNF compiler, the resulting target format can be reused for further analysis. For example, d-DNNFs and BDDs can be used to compute the number of satisfying assignments under assumptions (Darwiche 2001; Heß

et al. 2021), which could be used to compute the number of valid configurations containing a certain feature or an arbitrary combination of features (i.e., a partial configuration). Thus, if multiple #SAT computations are required, compiling into d-DNNF or BDD might be beneficial even if the compilation time takes longer than a DPLL-based computation.

5 Experiment design

In this section, we present the experiment design for our evaluation of #SAT solvers on industrial feature models. We provide the required information for the presentation (Section 6) and discussion (Section 7) of the results. The replication package for our empirical evaluation is publicly available.[2].

In Section 5.1, we explain the research questions we aim to answer in our evaluation. In Section 5.2, we present the gathered #SAT solvers and the methodology we used to collect them. In Section 5.3, we discuss the selection of subject feature models and provide information (e.g., number of features and domain) of the underlying product line. In Section 5.4, we describe the setup of the experiments regarding the overall procedure of the measurements, considered solvers, considered systems, and applied statistical tests. In Section 5.5, we provide details on the technical setup for the evaluation.

5.1 Research questions

In this section, we discuss the research questions that we aim to answer with the empirical evaluation. The research questions provide insight on the general scalability of #SAT technology, the performance of exact #SAT solvers and solver classes, the correlation between structural metrics of the feature model and the runtime of solvers, and the performance of approximate #SAT solvers. Typically, feature-model analyses, such as counting, are applied in interactive settings or in continuous integration. As those settings mandate short runtimes, we consider an analysis to be scalable if it requires at most a few minutes of runtimes.

RQ1 How do #SAT solvers perform on industrial feature models?

To use applications based on the feature-model cardinality in industry, we need to identify solvers that scale for the task of analyzing industrial feature models. Thus, we examine the performance of exact #SAT solvers regarding the runtime when computing the cardinality of industrial feature models. Furthermore, we aim to find the most efficient #SAT solvers to provide recommendations on which #SAT solvers to use. Here, we consider the runtime required to compute #SAT for given feature models as the efficiency of a solver.

RQ2 How do different classes of #SAT solvers perform on industrial feature models?

We consider multiple classes of #SAT solvers (cf. Section 5.3). With RQ2, we analyze the runtimes of different categories of solvers, namely (1) DPLLbased, (2) algebraic-based, and knowledge compilers translating to (3) BDD, (4) d-DNNF, and (5) other formats (i.e., EADT and SDD).We use the insights to discuss the benefits of different solver categories and to give recommendations on promising techniques for counting-based analyses.

RQ3 How does the runtime of #SAT solvers correlate to structural metrics of the feature model?

[2]https://doi.org/10.5281/zenodo.7329979

We aim to provide some first insights on which properties cause a feature model to be hard to analyze for #SAT solvers. In particular, we examine if there is a correlation between the performance of the #SAT solvers and structural metrics related to the size and complexity of feature models. The insights can be used as an indicator on the scalability of existing #SAT solvers for a given feature model depending on their structure. Furthermore, we identify metrics that have a high impact on performance to find promising candidates for more accurate performance predictions in the future. In Table 1, we provide a list of metrics we examined. For each metric, we provide a description and instructions on how to compute the respective metric for a given feature model. The metrics were collected by Bezerra et al (Bezerra et al. 2014) and Bagheri and Gasevic (Bagheri and Gasevic 2011). Those metrics are based on structural properties related to size (e.g., number of features) or related to complexity (e.g., cyclomatic complexity).

RQ4 How do approximate #SAT solvers perform on industrial feature models?

In addition to exact #SAT solvers, we examine the performance of approximate #SAT solvers which estimate the number of satisfying assignments for a given formula. There are applications which require exact results, such as uniform random sampling where approximate results would violate uniformity. However, for a multitude of applications, an estimated cardinality may be often sufficient. For example, consider prioritizing features that appear in many valid configurations for two features A and B with $\#A = 10^{65}$ and $\#B = 10^{60}$. For instance, an approximation that ensures that the result is at most 20 times larger/smaller than the exact count, would result in the same prioritizations as exact results as $\frac{1}{20} \cdot 10^{65} > 20 \cdot 10^{60}$. We examine the performance of approximate #SAT solvers to provide insights on their benefits.

5.2 Evaluated #SAT solvers

In the following, we present the #SAT solvers used in our empirical evaluation. First, we describe our methodology of gathering the solvers. Second, we list the identified solvers, group them by their type of computing #SAT, and provide pointers where to find them.

Methodology Our main goal for selecting the solvers is a representative coverage of publicly available #SAT solvers. Such a coverage should allow for conclusive results on (1) the current scalability of #SAT technologies when applied to feature models and (2) recommendations on which solvers should be used to analyze feature models at hand.

We included all solvers that satisfy the following criteria: First, the solver needs to be publicly available (i.e., source code or binary is provided at generally accessible URL). Second, the tools have to accept CNFs in DIMACS format as input. DIMACS is the de facto standard for representing CNFs and used by the vast majority of SAT and #SAT solvers (Biere 2008; Bayardo Jr and Pehoushek 2000; Sang et al. 2004, 2005; Darwiche 2002, 2004; Toda and Soh 2016). Third, the solver can be used as a standalone blackbox tool in contrast to tools that require further setup (e.g., a client-server architecture (Lagniez et al. 2018)). Furthermore, we excluded the tool GPUSAT (Fichte et al. 2019) which performs #SAT on a GPU as we expect issues with the comparability if a solver uses different hardware. We identified #SAT solvers with the following three approaches.

First, we collected work that performs counting-related analyses on product lines (Kübler et al. 2010; Munoz et al. 2019; Oh et al. 2016; Pérez Morago 2016; Heradio-Gil et al. 2011; Fernández-Amorós et al. 2014; Chen and Erwig 2011; Pohl et al. 2011) to identify #SAT tools and respective publications used in product-line analysis. Then, we performed

Table 1 Structural feature-model metrics (**RQ3**)

Number of features	
Description	Number of features in the feature model overall
Formula	$\|Features\|$, with $Features$ being the set of features
Number of leaf features	
Description	Number of features in the feature model without children
Formula	$\|Leaves\|$, with $Leaves$ being the set of leaf features
Number of top features	
Description	Number of children of the root feature
Formula	$\|Top\|$, with Top being the set of children of the root feature
Number of cross-tree constraints	
Description	Number of cross-tree constraints in the feature model
Formula	$\|CTC\|$, with CTC being the set of cross-tree constraints
Number of clauses	
Description	Number of clauses in the CNF representing the feature model
Formula	$\|Clauses\|$, with $Clauses$ being the set of clauses in the CNF
Number of literals	
Description	Number of overall literals appearing in the CNF
Formula	$\|Literals\|$, with $Literals$ being the set of literals in the CNF
CTC-Density	
Description	Ratio of unique features appearing in CTC compared to overall number of features
Formula	$\frac{\|Features_{CTC}\|}{\#\text{Features}}$, with $Features_{CTC}$ being the set of features appearing in a cross-tree constraint.
Depth of tree	
Description	Depth of the feature tree at the longest path
Formula	$\|Features_{LP}\|$, with $Features_{LP}$ being the set of features in the longest path from the root to a leaf feature.
Flexibility of configuration	
Description	Ratio of optional features compared to overall number of features
Formula	$\frac{\|Features_{Opt}\|}{\|Features\|}$ with $Features_{Opt}$ being the set of features that appear in some but not all valid configurations.
Ratio of variability	
Description	Average number of children
Formula	$\frac{\sum_{f \in Features} \|children_f\|}{\|Features \setminus Leaves\|}$, with $children_f$ being the set of children of feature f.
Coefficient of connectivity-density	
Description	Number of edges between features compared to the number of features
Formula	$\frac{\sum_{f \in Features} \|edges_f\|}{2 \cdot \|Features\|}$, with $edges_f$ being the set of distinct edges connecting features. For a clause $(f_1 \vee \ldots \vee f_n)$, there is an edge between every pair of feature $f_1, \ldots, f_n$.
Cyclomatic complexity	
Description	Number of distinct cycles between features
Formula	$\|cycles_f\|$ with $cycles_f$ being the set of independent cycles spanned by $edges_f$

(forward and backward) snowballing from the identified publications. For backward snowballing, we employed data from Google Scholar. Second, we used a list of publicly available #SAT solvers from the report of the model counting 2020 competition as comparison (Fichte

Table 2 Overview exact #SAT solvers

Solver	Type	Target format	Reference
`Cachet`	DPLL	–	(Sang et al. 2004, 2005)
`countAntom`	DPLL	–	(Burchard et al. 2015)
`Ganak`	DPLL	–	(Sharma et al. 2019)
`PicoSAT`	DPLL	–	(Biere 2008)
`Relsat`	DPLL	–	(Bayardo Jr and Pehoushek 2000)
`SharpCDCL`	DPLL	–	(Klebanov et al. 2013)
`sharpSAT`	DPLL	–	(Thurley 2006)
`McTW`	Algebraic	–	MC Competition (Fichte et al. 2021)
`SUMC1`	Algebraic	–	MC Competition (Fichte et al. 2021)
`ADDMC`	Algebraic	–	MC Competition (Fichte et al. 2021)
`c2d`	Compiler	d-DNNF	(Darwiche 2002; 2004)
`d4`	Compiler	d-DNNF	(Lagniez and Marquis 2017)
`dSharp`	Compiler	d-DNNF	(Muise et al. 2010)
`BuDDy`	Compiler	BDD	http://buddy.sourceforge.net/manual/main.html. Accessed: 02 Mar 2020
`CNF2OBDD`	Compiler	BDD	(Toda and Soh 2016)
`Cudd`	Compiler	BDD	https://github.com/vscosta/cudd. Accessed: 13 Jun 2020
`CNF2EADT`	Compiler	EADT	(Koriche et al. 2013)
`MiniC2D`	Compiler	SDD	(Oztok and Darwiche 2015)
`SDD`	Compiler	SDD	(Darwiche 2011)

et al. 2021). Both lists are similar with eleven shared #SAT solvers. The list of the model counting competition contained one solver in addition to the eleven shared solvers while the list from product-line analysis contained four additional solvers. Due to the similarity in the identified solvers, we argue that our list of #SAT solvers provides a reasonable representation of current #SAT technology. Third, we added three #SAT solvers that entered the model counting 2020 competition but were not published beforehand.[3]

Selected solvers Overall, we gathered 19 exact and 2 approximate #SAT solvers. Table 2 provides an overview on the exact solvers. The exact #SAT solvers can be separated in three main categories: DPLL-based solvers, algebraic solvers, and knowledge compilers. We consider a knowledge compiler to be a #SAT solver if the compiled target language and the compiler support computing the number of satisfying assignments in polynomial time in the size of the target formula.

The solver `countAntom` is the only solver which internally supports multi-threading (Burchard et al. 2015). We evaluated `countAntom` with one and four available threads separately to examine the impact of multi-threading on the runtime. We consider evaluating `countAntom` also with four threads (opposed to only with a single thread) as the more sensible option due to the following reasons: First, it is reasonable to assume that multiple threads would be used in industrial settings. Second, to allow multi-threading the developers

[3] https://doi.org/10.5281/zenodo.4292581

of `countAntom` made several adjustments which may put `countAntom` at a disadvantage when using a single thread. Third, nevertheless, a larger number of threads may result in a too large advantage for `countAntom`. During the remainder of the evaluation, we refer to `countAntom` with four threads if not stated otherwise.

Neither `BuDDy` http://buddy.sourceforge.net/manual/main.html. Accessed: 02 Mar 2020 nor `CUDD` https://github.com/vscosta/cudd. Accessed: 13 Jun 2020 support parsing DIMACS directly. In previous work, we implemented a Python-based wrapper called `ddueruem`[4] (Heß et al. 2021) which uses the `ctypes` library[5] to interface with their shared libraries and construct the BDD using the API described in their respective manuals http://buddy.sourceforge.net/manual/main.html. Accessed: 02 Mar 2020; https://github.com/vscosta/cudd. Accessed: 13 Jun 2020. As suggested in the manuals of `BuDDy` and `CUDD`, we enabled automatic variable reordering in both `BuDDy` and `Cudd`, using the converging variant of the sift algorithm (Rudell 1993). We decided to use our own wrapper `ddueruem` due to limitations, namely the missing support for `Cudd 3.0.0` and frequent crashes when using `BuDDy` in the JavaBDD[6] framework, which is often used for product-line analysis (Mendonça 2009; Mendonca and Cowan 2010; Pohl et al. 2011).

In addition to the exact #SAT solvers, we also identified two approximate #SAT solvers, namely `ApproxMC` (Chakraborty et al. 2013) and `ApproxCount` (Wei and Selman 2005). The solver `ApproxCount` iteratively assigns variables to reduce the complexity of a formula. For each assigned variable, the solver estimates the resulting reduction in the number of satisfying assignments. After a user-specified number of assigned variables, the exact #SAT solver `Cachet` is executed with the simplified formula as input. The estimated reduction is then applied to the result of the simplified formula to derive an approximated number of satisfying assignments for the original formula. In our previous work (Sundermann et al. 2020), every feature model with less than 1,000 features was successfully evaluated by most #SAT solvers. Following this insight, we directed `ApproxCount` to start the exact computation at 1,000 remaining variables. For `ApproxMC`, we used the default parameters.

5.3 Subject systems

The main goal of the empirical evaluation is to examine the applicability of #SAT solvers for analyzing feature models. We argue that the applicability mainly depends on the scalability on industrial feature models, as artificial models might not be representative for industrial usage as observed in other domains (Ansótegui et al. 2009; Heß et al. 2021). Therefore, we only use industrial feature models as subject systems. We consider a feature model to be industrial if it fulfills the following two criteria: (1) it specifies the variability of a product line used in the real world and (2) it does not vastly simplify the complexity (in terms of features and constraints) of the product line. Note that we only consider variability of the problem space (i.e., which valid configurations do exist) opposed to variability in the solution space (i.e., how and where to implement variability).

Selected systems With our selection of subject systems, we aim for a wide coverage of different domains. We evaluate the performance of the listed #SAT solvers on feature models taken from industrial product lines from the automotive, operating system, database, and

[4] https://github.com/SoftVarE-Group/emse-evaluation-sharpsat/tree/v1.0/solvers/ddueruem
[5] https://docs.python.org/3/library/ctypes.html
[6] http://javabdd.sourceforge.net/

Table 3 Overview Subject Systems (Sorted by #Features)

Subject systems	i	#Features	#Constraints	Orig. source
BerkeleyDB	1	76	20	https://github.com/FeatureIDE/FeatureIDE
axTLS	2	96	14	(Knüppel et al. 2017)
uClibc	3	313	56	(Knüppel et al. 2017)
uClinux-base	4	380	3,455	(Knüppel et al. 2017)
Automotive04	5	531	623	Confidential
Automotive03	6	588	1,184	Confidential
BusyBox	7	631	681	(Pett et al. 2021)
FinancialServices	8	771	1,080	(Nieke et al. 2018; Pett et al. 2019)
Embtoolkit	9	1,179	323	(Knüppel et al. 2017)
CDL (116 Models)	10	1,178–1,408	816–956	(Knüppel et al. 2017)
uClinux-distribution	11	1,580	197	(Knüppel et al. 2017)
Automotive05	12	1,663	10,321	Confidential
Automotive01	13	2,513	2,833	https://github.com/FeatureIDE/FeatureIDE
Linuxv2.6.33.3	14	6,467	3,545	(Knüppel et al. 2017)
Automotive02	15	18,616	1,369	(Knüppel et al. 2017)

financial services domain. Table 3 provides an overview on the considered feature models, sorted by the number of features, including name, number of features, number of constraints, and the work they were originally published in. The index *i* indicates the position of the subject system in diagrams in Section 6.

First, we analyze feature models provided by Knüppel et al (Knüppel et al. 2017).[7] The authors extracted the systems from snapshots of an automotive product line and by translating KConfig and CDL models. KConfig[8] is a language designed for managing Linux configurations and CDL for managing eCos[9], a configurable operating system for embedded applications (Knüppel et al. 2017). The considered KConfig models are *axTLS*, *uClibc*, *uClinux-base*, *Embtoolkit*, *uClinux-distribution*, and *Linux*. In addition, Knüppel et al provide an automotive product line *Automotive02*. Second, we evaluate the solvers on *BusyBox* provided by Pett et al (Pett et al. 2021).[10] Third, we include a feature model from the *FinancialServices* domain (Nieke et al. 2018; Pett et al. 2019). Fourth, we consider the systems *Automotive01* (Kowal et al. 2016) and *BerkeleyDB* (Kästner et al. 2007) which are available as FeatureIDE examples.[11] Fourth, we were given access to industrial models for three different systems from the automotive domain (*Automotive03-Automotive05*). These models were provided in a proprietary format. With the help of company interns, we translated their configuration knowledge into feature models. For our entire experiment, we translated each feature model to the DIMACS format using FeatureIDE 3.5.5.[12]

For some subject systems, namely *Automotive02–05*, *FinancialServices*, and *BusyBox* a history of feature models is available, each representing a unique timestamp. For each

[7] https://github.com/AlexanderKnueppel/is-there-a-mismatch
[8] https://www.kernel.org/doc/html/latest/kbuild/kconfig-language.html
[9] https://ecos.sourceware.org/
[10] https://github.com/TUBS-ISF/Stability-of-Productline-Sampling
[11] https://github.com/FeatureIDE/FeatureIDE
[12] https://github.com/FeatureIDE/FeatureIDE/releases/tag/v3.5.5

feature model with a history, we consider only the latest version. Thoroughly analyzing the entire history is out of scope and left as future work.

In previous experiments (Sundermann et al. 2020), we found that the 116 CDL models are highly similar regarding a variety of metrics (i.e., all metrics considered Section 5.1) and also resulted in similar runtimes for #SAT solvers. If we evaluated the different CDL models as distinct systems, 89.2% of the overall 130 (14 other + 116 CDL) evaluated feature models are CDL models which results in a huge bias of the results. Therefore, we consider the median of runtimes over the 116 different CDL models as the runtime of the CDL subject system in every experiment if not stated otherwise. To show the similarity in the performance of #SAT solvers, we also present the runtimes on the different CDL models in Section 6.

5.4 Experimental setup

In this section, we describe the procedure of the experiments conducted to gather insights to answer our research questions.

To analyze the feature models with #SAT solvers, we translate each feature model into conjunctive normal form (CNF) and store it in DIMACS format. We invoke the #SAT solvers with the DIMACS as input. As the translation to CNF typically requires only a few milliseconds and is equivalent for each solver, we do not include the translation time in the overall runtime. Furthermore, we set a timeout of ten minutes (cf. **RQ1** in Section 5.1) for evaluating a single feature model as the baseline for the experiment. The threshold is motivated by applying counting-based analyses in interactive settings and continuous-integration environment which should not exceed a few minutes of runtime to provide fast feedback to developers after changing a feature model.

An important aspect we consider for our benchmark is the trade-off between significance of results and ecological footprint. If a solver hits the timeout of ten minutes for every single one of the 130 feature models, the evaluation would require almost a day of continuous runtime considering a single repetition. Performing a number of repetitions that allows for significant results would require substantially more runtime. For instance, when using 50 repetitions this would potentially result in more than 11 years of nonstop computation time. Thus, we aim to reduce the overall runtime of the experiments while preserving significant results.

In the following, we explain the performed experiments in detail. Table 4 provides an overview over the two experiments regarding considered solvers, considered research questions, and number of performed repetitions per measurement.

Table 4 Overview experiments

Experiment	Solvers	#Reps.	RQ1	RQ2	RQ3	RQ4
Experiment 1a	All Exact #SAT	1	×	×		
Experiment 1b	Remaining Exact #SAT	50	×	×	×	
Experiment 1c	Remaining Exact #SAT	1	×	×		
Experiment 2a	All Approximate #SAT	1				×
Experiment 2b	Remaining Approximate #SAT	50				×

Experiment 1: Scalability of exact #SAT solvers In the first experiment, we measure the runtimes of the exact #SAT solvers (cf. Section 5.2) on the considered feature models (cf. Section 5.3). For the solvers based on knowledge compilation, we consider the overall runtime to compile and compute a result. We use the insights of this experiment to answer **RQ1**, **RQ2**, and **RQ3**. The experiment is separated into three stages.

In the first stage (Experiment 1a), we identify and filter slow #SAT solvers. Here, we measure the runtime of each of the 19 exact #SAT solvers on the 15 subject systems. The idea of Experiment 1a is to remove slow solvers from the following two stages that significantly increase the overall runtime for the experiments. We consider a solver to be slow if the solver requires more than 50% of additional runtime compared to the following solver when ordered by overall runtime for the experiment. We refer to the solvers that are not excluded as *remaining* solvers. In the second stage (Experiment 1b), we perform the measurements with the remaining #SAT solvers with 50 repetitions for each feature model for more robust results. In the third stage (Experiment 1c), we further evaluate the runtimes of the remaining solvers on subject systems for which no solver computed a result in Experiment 1a and 1b. Here, we perform one repetition and increase the timeout to 24 hours to examine whether an increase of the timeout allows a successful computation.

Orthogonal to the measurements of Experiment 1a, 1b, and 1c, we examine the number of valid configurations for the considered feature models. Here, we use the results computed by the solvers.

Experiment 2: Scalability of approximate #SAT solvers In the second experiment, we examine the scalability of the two considered approximate #SAT solvers on each feature model to provide insights for **RQ4**. Furthermore, we give a comparison to the exact #SAT solvers to evaluate the benefits. Analogous to Experiment 1a, we also perform an initial experiment referred to as Experiment 2a with only one repetition per measurement to exclude slow solvers from the following experiments. Then, we repeat the measurements with 50 repetitions on the remaining approximate #SAT solvers (Experiment 2b), analogous to Experiment 1b.

Statistical tests We apply the following statistical tests to evaluate the significance of our results depending on the use case. Table 5 gives an overview on the use cases, tests we used for each use case, and the **RQs** that are dependent on the given use case.

For the first use case *Comparison Solvers System*, we compare the performance of different solvers on each of the feature models separately. For the comparison, we consider the 50 repetitions of a solver/system combination as sample. Here, we apply a Mann-Whitney significance test (McKnight and Najab 2010) as we have *unpaired* samples and do not assume

Table 5 Overview statistical tests

Use case	Sample	Statistical tests	RQ1	RQ2	RQ3	RQ4
Comparison solvers system	Unpaired	Mann-Whitney (McKnight and Najab 2010)	×	×		×
Comparison solvers overall	Paired	Friedman test (Conover and Iman 1981)	×	×		×
		Post-Hoc Conover (Conover and Iman 1981)				
Correlation solver/metric	Paired	Spearman (Zar 1972)			×	

Table 6 Spearman: levels of correlation

Correlation	Value range
Very weak	$0 \leq \lvert r_s \rvert < 0.2$
Weak	$0.2 \leq \lvert r_s \rvert < 0.4$
Moderate	$0.4 \leq \lvert r_s \rvert < 0.6$
Strong	$0.6 \leq \lvert r_s \rvert < 0.8$
Very strong	$0.8 \leq \lvert r_s \rvert \leq 1.0$

a normal distribution. For the tests, we assume the typical significance level of $\alpha = 5\%$. We use the `scipy`v1.7.2 implementation of Mann-Whitney.[13]

For the second use case *Comparison Solvers Overall*, we compare the overall performance of the different #SAT solvers on all 15 subject systems. For the comparison, each data point corresponds to the median of runtimes over the 50 repetitions for a combination of subject system and solver. Here, we apply a Friedman Test followed by a Post-hoc Conover test on the samples of all solvers as we have *paired* samples (pairs of subject systems) and again do not assume a normal distribution (Conover and Iman 1981). For the tests, we assume a significance level $\alpha = 5\%$. We use the `scipy`v1.7.2 implementation of Friedman[14] and the `scikit_posthocs` implementation of Post-Hoc Conover.[15]

For the third use case *Correlation Solver/Metric*, we evaluate the correlation between the runtime of #SAT solvers and structural metrics (cf. **RQ3**). Here, we use Spearman's correlation coefficient r_s (Zar 1972) to evaluate the strength of the correlation. For two variables, r_s describes their correlation with a value between -1 and 1. The values 1 and -1 describes a very strong positive or negative correlation, respectively. $r_s = 0$ indicates that the variables have no correlation at all. We expect that Spearman's coefficient provides more sensitive results (compared to Pearson's coefficient) due to the following reasons (Artusi et al. 2002). First, Spearman's coefficient is suitable to detect non-linear relationships between the variables, and it is possible that the correlation between a metric and the runtimes is not linear. Second, there may be significant outliers which tend to cause problems for the expressiveness of Pearson's coefficient. Table 6 shows the levels of correlation strength for the Spearman's coefficient r_s we use. We use the `scipy`v1.7.2 implementation of Spearman to compute the correlation coefficients[16].

In addition to significance tests, we compute effect sizes (Sullivan and Feinn 2012) for samples shown to be significantly different. In particular, we employ Cohen's d which describes the difference between the median of two samples relative to the standard deviation (Sullivan and Feinn 2012). Table 7 shows the levels of effect sizes with their range of d values (Sullivan and Feinn 2012).

5.5 Technical setup

Each experiment was performed on a *Linux CentOS 8* system with *64*-bit architecture. The evaluated machine uses an *Intel Core Broadwell Processor* that consists of *16* sockets with

[13] https://docs.scipy.org/doc/scipy/reference/generated/scipy.stats.mannwhitneyu.html
[14] https://docs.scipy.org/doc/scipy/reference/generated/scipy.stats.friedmanchisquare.html
[15] https://scikit-posthocs.readthedocs.io/en/latest/generated/scikit_posthocs.posthoc_conover/
[16] https://docs.scipy.org/doc/scipy/reference/generated/scipy.stats.spearmanr.html

Table 7 Cohen: levels of effect size

Effect size	Value range
Very small	$0 \leq d < 0.2$
Small	$0.2 \leq d < 0.5$
Medium	$0.5 \leq d < 0.8$
Large	$0.8 \leq d < 1.3$
Very large	$1.3 \leq d$

one core each. The clock rate is *2,394 Mhz* and the machine contains *62 GB* of RAM. For each computation and experiment, we limit the memory usage to 8 GB due to the following two reasons. First, we assume that a memory limit of 8 GB reflects the capacity for RAM usage on common PCs or notebooks. Second, in preliminary experiments, we found that further increasing the memory limit yields little to no benefits for the runtimes of #SAT solvers. For the measurements, we implemented a *Python* framework to (1) call the solver binaries and provide the input, (2) measure the runtimes with the *timeit* module,[17] and (3) limit the memory usage. For reproducibility, the framework, solvers, and input data are publicly available.[18] *Hyper-threading*, *turboboost*, and *caching of the file system* were disabled during the entire measurements to reduce computational bias. Furthermore, no other major computations were run on the system during the experiments.

6 Results

In this section, we present the results of our empirical evaluation separated into the two presented experiments.

6.1 Experiment one: Exact #SAT solvers

Experiment 1a Figure 2 shows the runtime of all exact #SAT solvers on each subject system. Each point on the x-axis corresponds to one of the 15 subject systems. The systems are sorted by the number of features in ascending order (cf. Table 3). The y-axis shows the runtime of the different solvers with a logarithmic scale. The different categories are indicated by the colors of the markers (orange = DPLL, purple = algebraic, green = d-DNNF, blue = BDD, gray = knowledge compilers to other formats). The red line indicates that a solver hits the timeout. The blue line indicates that an error occurred or a solver passed the memory limit. *CDL Median* corresponds to the median over all 116 CDL feature models (cf. Section 5.3). The majority of systems (13/15) was successfully evaluated within 10 minutes by at least one solver. For each of the 13 solved systems, the fastest solver required less than one second. However, none of the solvers was able to compute the cardinality of the other two systems, namely *Automotive05* and *Linux*.

Figure 3 shows the sum of runtimes of each evaluated exact #SAT solver on all 15 subject systems in Experiment 1a. Each bar corresponds to the sum of runtimes for one solver. Note that this sum only includes the median of runtimes for the 116 CDL models instead of the

[17] https://docs.python.org/3/library/timeit.html
[18] https://doi.org/10.5281/zenodo.7329979

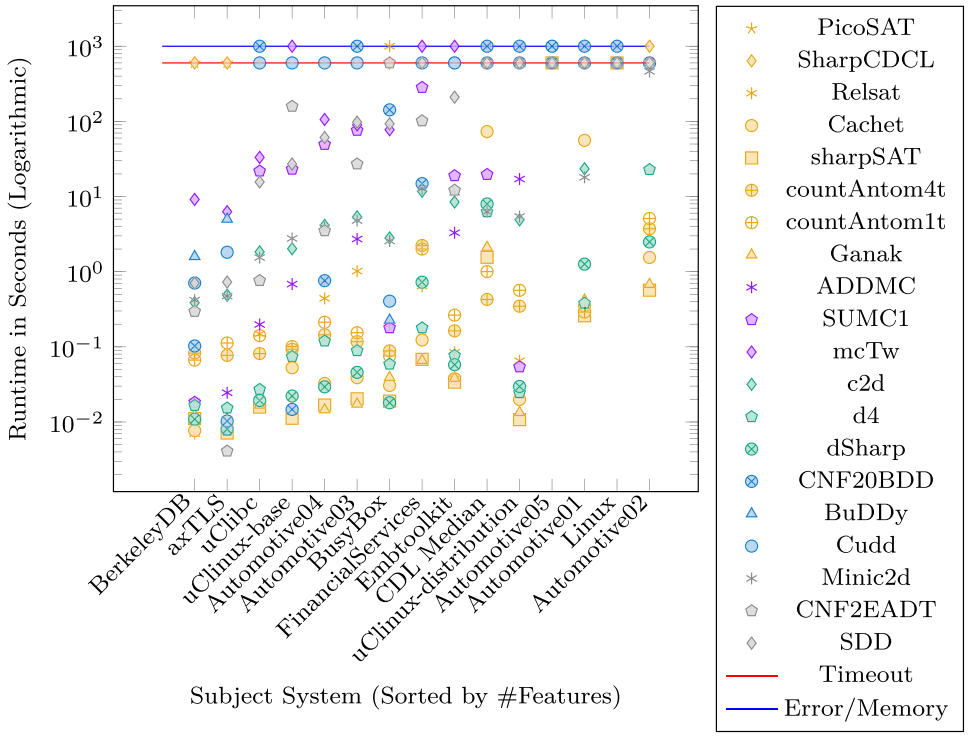


Fig. 2 Runtime in seconds for all exact #SAT solvers

overall sum (cf. Section 5.3). Considering a timeout of 10 minutes per system, the maximum runtime is 150 minutes (hitting the timeout for all 15 systems) which is indicated by the red line. The solvers are sorted by the overall sum of runtimes (ascending). If a subject system could not be evaluated due to timeout, memory limit, or an arbitrary error, we added the timeout (10 minutes) to the overall runtime. Table 8 gives an overview of the performance

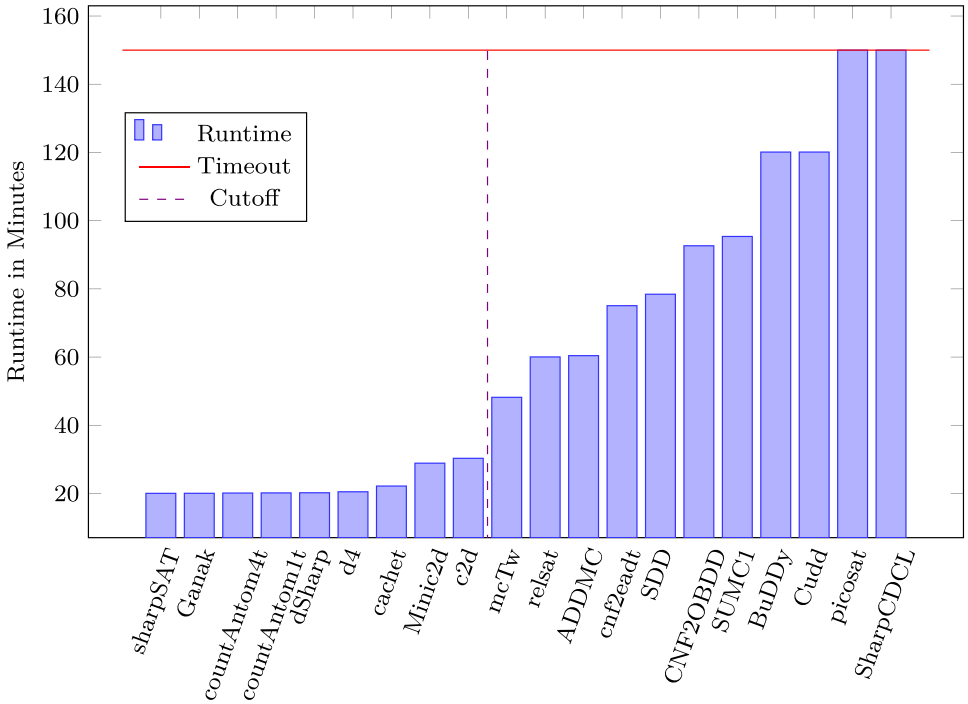


Fig. 3 Runtime of all solvers to evaluate the 15 subject systems

Table 8 Overview Experiment 1a

Solver	Solved	% Solved	Ov. Runtime (s)	Remaining
sharpSAT	13	87	1,202.6	✓
Ganak	13	87	1,203.4	✓
countAntom4t	13	87	1,207.8	✓
countAntom1t	13	87	1,210.2	✓
dSharp	13	87	1,212.7	✓
d4	13	87	1,230.1	✓
Cachet	13	87	1,339.9	✓
miniC2D	13	87	1,733.6	✓
c2d	13	87	1,818.9	✓
mcTw	11	73	2,892.0	✗
Relsat	9	60	3,602.4	✗
ADDMC	9	60	3,625.0	✗
CNF2EADT	8	53	4,504.1	✗
SDD	8	53	4,705.7	✗
CNF2OBDD	6	33	5,558.1	✗
SUMC1	6	33	5,721.8	✗
BuDDy	3	20	7,206.6	✗
Cudd	3	20	7,206.8	✗
SharpCDCL	0	0	9,000.0	✗
PicoSAT	0	0	9,000.0	✗

for the different solvers. Eight of the solvers, namely `sharpSAT`, `Ganak`, `countAntom` (both with four and one thread), `d4`, `Cachet`, `dSharp`, `MiniC2D`, and `c2d` evaluated 13 out of 15 (86.7%) subject systems within eleven minutes of runtime. The eleven slower solvers, namely `McTW`, `Relsat`, `ADDMC`, `CNF2EADT`, `SDD`, `CNF2OBDD`, `SUMC1`, `BuDDy`, `Cudd`, `SharpCDCL`, and `PicoSAT`, successfully evaluated at most 73.3% of the 15 subject systems with a timeout of ten minutes for each model. Furthermore, the fastest of the slower solvers (`McTW`) requires around 60% more runtime than the slowest of the faster solvers (`c2d`). Overall, the eleven slower solvers took 96.8% of the total runtime (10.2 days) required for Experiment 1a. Performing the 50 repetitions with all solvers would result in around 1.4 years of continuous computation just for Experiment 1b. For all following experiments, we only include the eight fastest #SAT solvers. In Table 8 the excluded solvers are marked with an ✗-mark in the column *remaining*. In Fig. 3, the dashed violet line marks the cut for the excluded solvers. Each solver on right side of the line is excluded from the following experiments.

Each BDD-based #SAT solver successfully evaluated at most 6 of the 15 systems and required at least 92 minutes overall. Every d-DNNF-based solver needed less than 31 minutes for all subject systems and only failed to evaluate *Linux* and *Automotive05*.

Figure 4 shows the runtime of the 19 exact #SAT solvers for the 116 CDL feature models. Each point on the x-axis corresponds to one CDL model. The y-axis shows the runtime of different solvers in seconds with a logarithmic scale. For all solvers but `Cachet`, the median runtime over all 116 feature models is smaller than two times the minimum value (i.e., the shortest runtime required for one of the 116 feature models for that solver). Also,

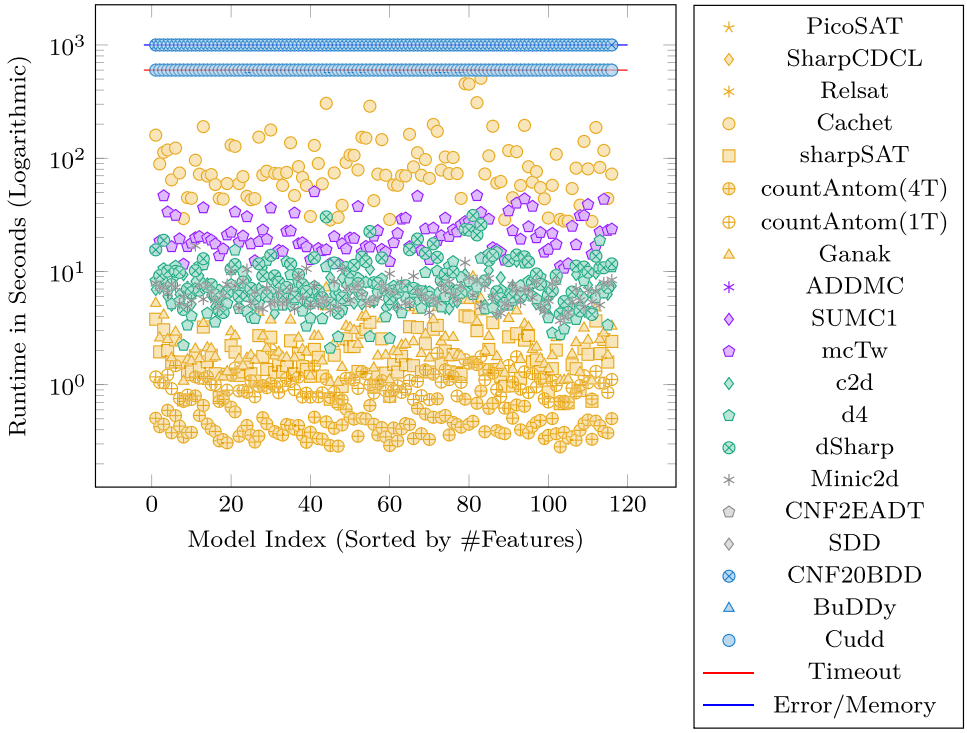


Fig. 4 Runtime in seconds for all exact #SAT solvers on CDL models

the maximum is always smaller than two times the median. The results support our claim in Section 5.3 that CDL models are highly similar and handling them as separate subject systems would result in a bias of the measured runtimes.

Experiment 1b In Experiment 1b, we measured each combination of the feature models and the eight remaining solvers with 50 repetitions for more reliable results (cf. Section 5.4). Figure 5 shows the median runtimes and standard deviation for each solver-system combination. In the remainder of this section, we only consider the 13 systems successfully evaluated by at least one of the solvers if not stated otherwise. Considering the overall sum of runtimes, the three solvers requiring the least runtime are `sharpSAT` (2.5 seconds), `Ganak` (3.3 seconds), and `countAntom` (7.8 seconds). Over the 13 systems, `sharpSAT` is significantly ($p < 0.03$) faster than every #SAT solver but `Ganak`, `Cachet`, and `dSharp`. However, each effect size is small ($d < 0.47$) which matches the expectations as the large differences in runtime between the subject systems for each solver result in a large standard deviation.

`sharpSAT` is significantly ($p < 0.004$) faster with mostly (88.1%) very large effect sizes ($d > 1.53$) than all other solvers on 6 of the 13 systems. `Ganak` is significantly ($p < 10^{-11}$) faster than all other #SAT solvers with very large effect sizes ($d > 1.61$) for *Automotive03*. `countAntom` is significantly faster ($p < 10^{-17}$) than all other solvers with very large effect sizes ($d > 1.73$) on all 116 *CDL* models but for no other system. `Cachet` is significantly ($p < 10^{-7}$) faster than all other solvers for three smaller (less than 1,200 features) systems, namely *axTLS*, *embToolkit*, and *BerkeleyDB* with all effect sizes being very large ($d > 1.38$) but one with a medium effect size ($d = 0.78$).

We also compared `countAntom` with four and one thread. `countAntom` with four threads is significantly ($p = 0.028$) faster than with one thread for the 13 systems overall. Still, `countAntom` with one thread is significantly ($p < 0.036$) faster for three subject systems, namely *embtoolkit* ($d = 0.60$), *uClinux-base* ($d = 0.82$), and *Automotive03*

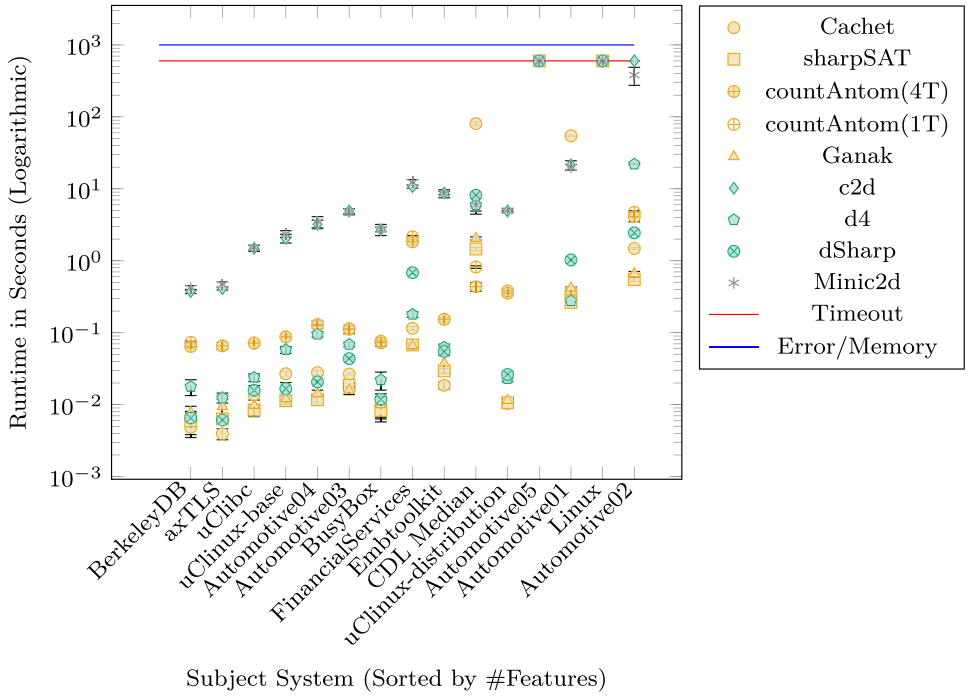


Fig. 5 Runtime (median & standard deviation) in seconds for remaining exact #SAT solvers

($d = 0.21$). Overall, `countAntom` with four and one thread require 7.8 and 9.2 seconds of runtime, respectively.

Table 9 shows the correlation between structural metrics of the feature models and the runtime of #SAT solvers. 'Correlation Fastest' shows the correlation between each metric and the runtime of the fastest solver for each instance. Note that the fastest solver varies depending on the evaluated feature model. There is a very strong positive ($r_s > 0.8$) correlation between the runtime and the following metrics: number of features, number of leaf features, number of cross-tree constraints, number of literals, and number of clauses. Consequently, for instance, the runtime of #SAT solvers tends to increase if the number of features increases. Also, there is a strong correlation between the cyclomatic complexity and the runtime. Every other metric correlates either weakly (0.2–0.39) or very weakly ($r_s < 0.2$) with the runtime of the fastest solver. 'Correlation Range' shows the minimum and maximum correlation between a metric and a solver. For every metric that has a strong correlation with the runtime of the fastest solver, each solver has an at least strong correlation. This observation is analogous for weakly correlated metrics with one exception (`countAntom` has a moderate correlation with the connectivity density).

Figures 6 and 7 show the runtime of the eight remaining solvers in relation to the number of features and the number of constraints, respectively. In each diagram, both scales are logarithmic. Every system with either fewer than 1,000 features or 1,000 constraints was evaluated within 0.5 seconds. While there is a strong correlation between the runtimes of the #SAT solvers and both metrics (i.e., number of features and constraints), a feature model with respectively more features or constraints does not guarantee a longer runtime. The two systems that reached the timeout, namely *Linux* and *Automotive05*, contain 6,467 and 1,663 features. *Automotive02* which contains 18,616 features was evaluated within 0.5 seconds. It is important to note that *Automotive02* contains only 1,369 constraints while *Linux* and *Automotive05* contain 3,545 and 10,321 constraints, respectively. Also, *uClinux-base* contains 3,455 constraints, but the fastest solver is about 50 times faster than for *Automotive02*.

Table 9 Correlation between structural metrics and runtime of #SAT solvers

Metric	Coefficient fastest	Coefficient range
Number of literals	0.89 (very strong)	0.82 (very strong)–0.89 (very strong)
Number of clauses	0.87 (very strong)	0.80 (very strong)–0.87 (very strong)
Number of features	0.85 (very strong)	0.73 (strong) –0.94 (very strong)
Number of leaf features	0.82 (very strong)	0.68 (strong) –0.92 (very strong)
Number of constraints	0.81 (very strong)	0.67 (strong)–0.84 (very strong)
Cyclomatic complexity	0.77 (strong)	0.64 (strong)–0.79 (strong)
Tree depth	0.34 (weak)	0.29 (weak)–0.39 (weak)
Connectivity density	0.20 (weak)	0.11 (very weak)–0.57 (moderate)
Ratio of variability	0.11 (very weak)	-0.01 (very weak)–0.27 (weak)
Number of top features	0.06 (very weak)	-0.01 (very weak)–0.17 (very weak)
Flexibility of configuration	0.07 (very weak)	-0.01 (very weak)–0.18 (very weak)

Experiment 1c In Experiment 1c, we invoked the remaining solvers with a timeout of 24 hours for the two systems which hit the timeout for every solver in every repetition, namely *Automotive05* and *Linux*. Neither of the remaining eight solvers was able to compute the cardinality for either system within 24 hours.

Feature-model cardinalities Table 10 shows the cardinalities (i.e., the number of valid configurations) of the evaluated subject systems. The systems are sorted by their number of features. Note that the computed cardinalities are equal for all solvers. For *Linux* and *Automotive05*, the cardinality is unknown as no solver was able to compute a result. For the remaining systems, the cardinality ranges from $4.1 \cdot 10^9$ (*BerkeleyDB*) to $1.7 \cdot 10^{1534}$ (*Automotive02*).

Figure 8 shows the cardinality of the 13 successfully evaluated subject systems in relation to their number of features. There is a very weak positive correlation between the number of features and the cardinality (0.03 with Spearman). In several cases, a feature model with

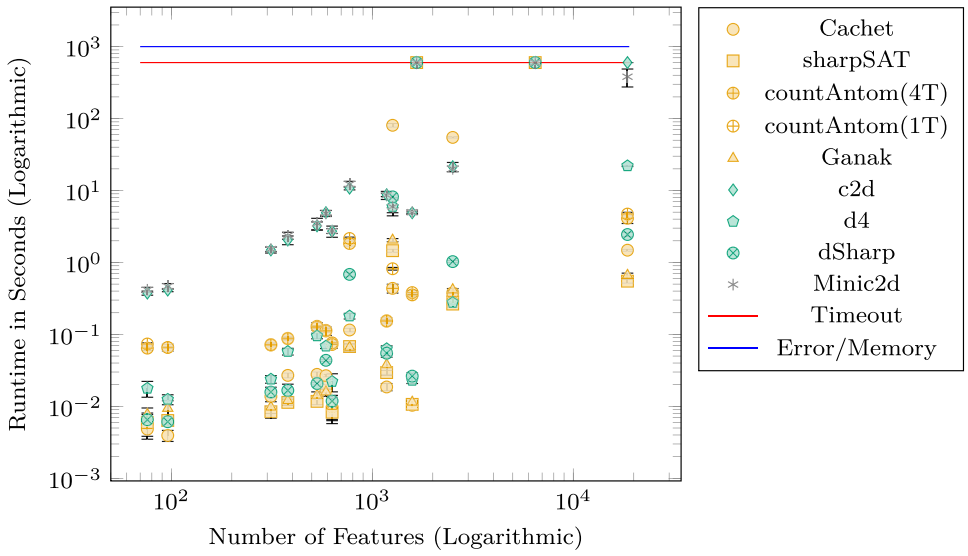


Fig. 6 Runtime of solvers in relation to the number of features

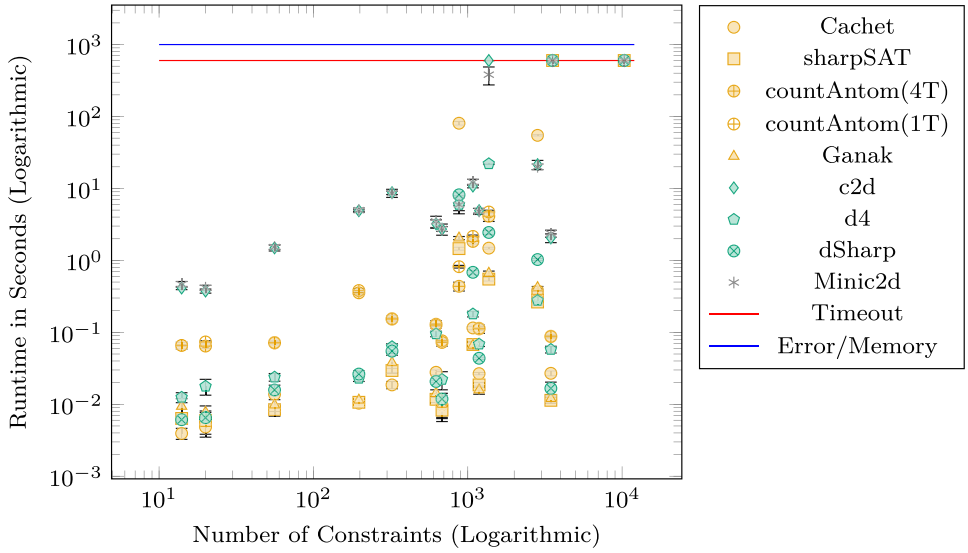


Fig. 7 Runtime of solvers in relation to the number of constraints

fewer features has a higher cardinality. For instance, *BusyBox* has 631 features and a cardinality of $2.1 \cdot 10^{201}$ while *FinancialServices* has 771 features and a cardinality of $9.7 \cdot 10^{13}$. Still, the three feature models with the largest number of features also have the highest cardinality. For instance, *Automotive02* has by far the largest cardinality $1.7 \cdot 10^{1534}$ and also seven times more features than *Automotive01* which has the second-highest number of features (disregarding Linux as we do not know its cardinality).

6.2 Experiment two: approximate #SAT solvers

Experiment 2a Figure 9 shows the runtimes of both evaluated approximate #SAT solvers on each feature model. `ApproxMC` hit the timeout of ten minutes for each but the two

Table 10 Cardinalities of Subject Systems (Sorted by #Features)

Subject systems	Number of valid configurations
BerkeleyDB	$4.1 \cdot 10^{9}$
axTLS	$8.3 \cdot 10^{11}$
uClibc	$1.7 \cdot 10^{40}$
uClinux-base	$2.6 \cdot 10^{22}$
Automotive04	$2.5 \cdot 10^{21}$
Automotive03	$2.5 \cdot 10^{31}$
BusyBox	$2.1 \cdot 10^{201}$
FinancialServices	$9.7 \cdot 10^{13}$
Embtoolkit	$5.1 \cdot 10^{96}$
CDL (116 Models)	$2.6 \cdot 10^{118} - 3.0 \cdot 10^{136}$
uClinux-distribution	$4.1 \cdot 10^{409}$
Automotive05	unknown
Automotive01	$5.4 \cdot 10^{217}$
Linux	unknown
Automotive02	$1.7 \cdot 10^{1534}$

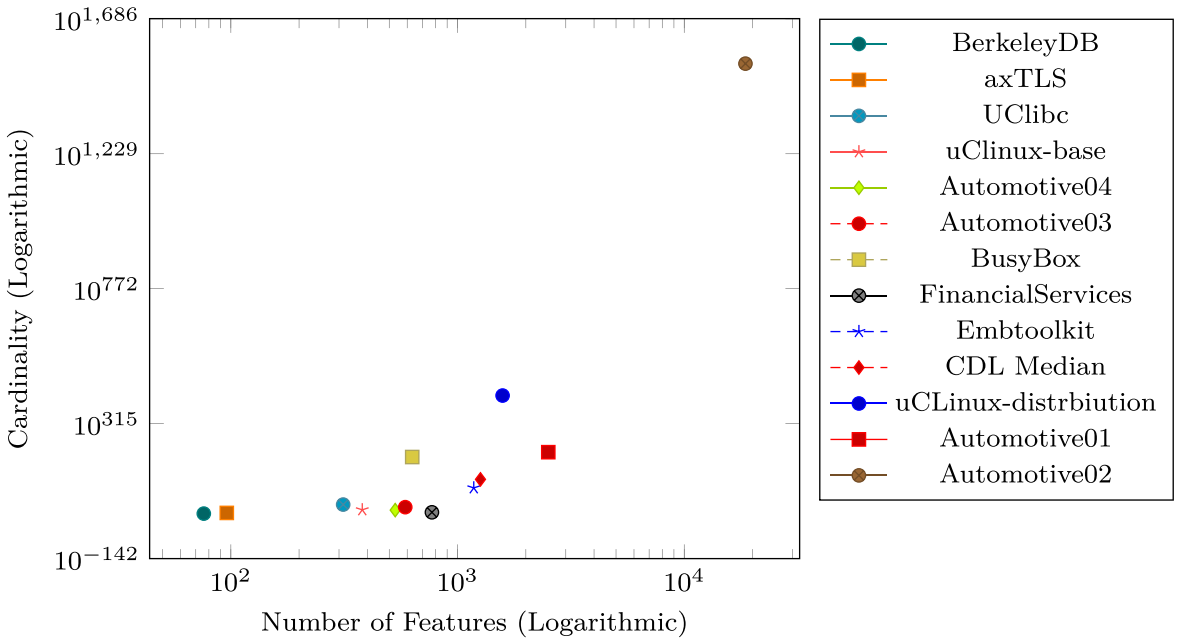


Fig. 8 Cardinality of subject systems in relation to number of features

smallest models, namely *axTLS* (96 features) and *BerkeleyDB* (76 features). Consequently, `ApproxMC` was excluded for *Experiment 2b*. `ApproxCount` hit the timeout for four subject systems.

Experiment 2b Figure 10 shows the runtimes of the best performing approximate #SAT solver (`ApproxCount`) with the exact #SAT solver that required the least time overall, namely `sharpSAT`. `ApproxCount` hit the timeout for four subject systems, while

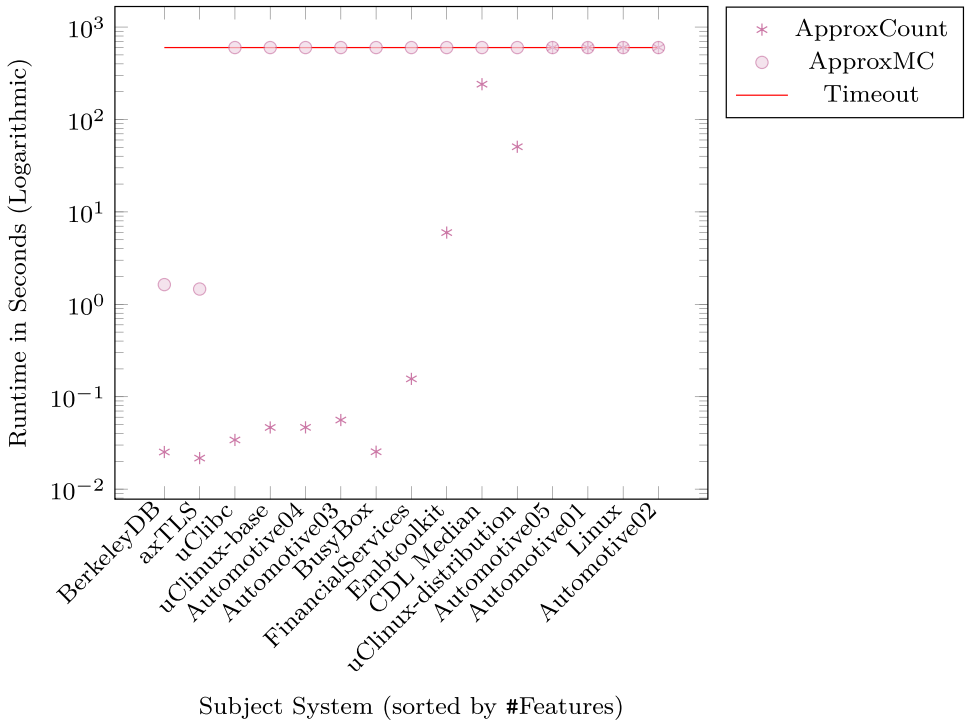


Fig. 9 Runtime in seconds for approximate #SAT solvers

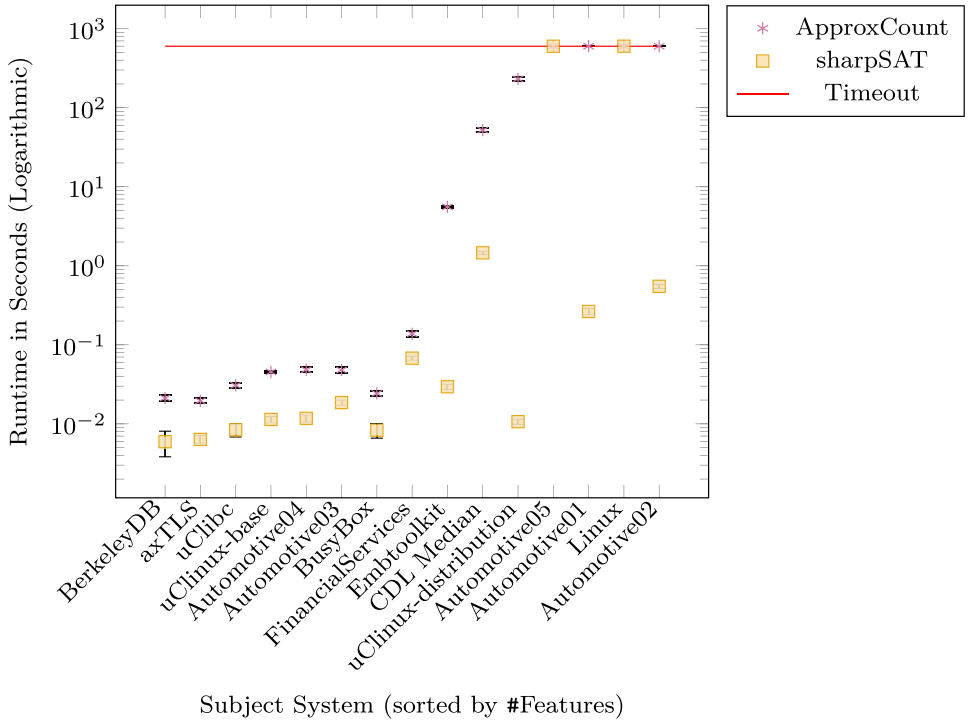


Fig. 10 Comparison runtime (median & standard deviation) ApproxCount vs sharpSAT

`sharpSAT` hit the timeout for two subject systems. For all 13 feature models that were successfully evaluated by at least one solver, `sharpSAT` is significantly faster than `ApproxCount` ($p < 0.0002$) with very large ($d > 5.1$) effect sizes for every single feature model. Furthermore, `ApproxCount` needs 5 minutes for 11 out of 15 systems while `sharpSAT` requires less than 2 seconds for those.

7 Discussion

In this section, we discuss the results regarding our research questions.

RQ1 *How do #SAT solvers perform on industrial feature models?* Our results indicate that the scalability of the #SAT solvers depends on the evaluated feature model. Based on our results, we expect that most industrial feature models can be evaluated within minutes or even seconds by the faster #SAT solvers we identified. Overall, 13 of the 15 analyzed feature models were successfully evaluated within 10 minutes. In addition, the fastest solver for each of those feature models required even less than one second which we consider scalable as it satisfies typical time restrictions of interactive environments and continuous integration environments. Nevertheless, there are systems for which no available #SAT solver scales. In our experiment, two systems, namely *Automotive05* and *Linux*, could not be evaluated by any solver not even within a timeout of 24 hours. Our results indicate that the hardness of both systems lies in their high number of features and constraints (c.f. the answer for **RQ3**).

Eight solvers, namely `sharpSAT`, `Ganak`, `countAntom`, `dSharp`, `d4`, `Cachet`, `MiniC2D`, and `c2d`, successfully evaluated 13 of 15 systems within 10 minutes of overall runtime. `sharpSAT` requires the least time to evaluate the 13 subject systems (2.6

seconds overall) closely followed by `Ganak` (3.4 seconds). While some solvers performed overall better than others, none of the solver is superior to the other solvers on every feature model. The results indicate that some solvers are inferior regarding the task of computing the cardinality of feature models, namely `PicoSAT`, `Relsat`, `SharpCDCL`, `McTW`, `SUMC1`, `CNF2OBDD`, `BuDDy`, `Cudd`, `CNF2EADT`, and `SDD`. Those solvers hit the timeout for at least four subject systems and some even for all systems while being substantially slower for the systems they successfully evaluated.

RQ2 *How do different classes of #SAT solvers perform on industrial feature models?* For single #SAT invocations, as performed in the experiment design at hand, we recommend the usage of the fastest DPLL-based solvers. The three best performing solvers, namely `sharpSAT`, `Ganak`, and `countAntom` are based on exhaustive DPLL.

For multiple #SAT invocations, reusing d-DNNFs seems promising. All d-DNNF compilers are part of the eight fastest solvers. For each feature model, that was successfully evaluated by at least one solver, the fastest d-DNNF-based solvers `dSharp` and `d4` require at most a few seconds in sum for compilation and model counting. For each follow-up computation, the compiled d-DNNF could be re-used (e.g., for computing the number of valid configurations containing certain features). Hence, we expect d-DNNF solvers are likely faster when performing multiple computations, which is required for the majority of counting-based analyses (Sundermann et al. 2021). SDDs can also be re-used and, thus, are a considerable candidate but the best performing SDD-based solver (`MiniC2D`) was substantially slower than `dSharp` (42 times slower) and `d4` (18 times slower).

The remaining types of #SAT solvers, namely algebraic-based, BDDs, and other knowledge compilation formats performed substantially worse than the eight fastest solvers. Both algebraic solvers, namely `ADDMC` and `SUMC1`, overall successfully evaluated only nine (60%) of the subject systems. Hence, we excluded both solvers after Experiment 1a, and we cannot recommend using these solvers for #SAT-based analysis of feature models. The three BDD libraries overall successfully evaluated only six (40%) subject systems. `BuDDy` and `Cudd` even hit the timeout for 12 of 15 subject systems. Therefore, we do not recommend to use current BDD libraries for computing the cardinality of feature models. Nevertheless, BDDs are tractable (i.e., have polynomial time complexity w.r.t. to the size of the BDD) for additional computation types, such as existential quantification (Bryant 2018). Using BDDs for other feature-model analyses may thus still be beneficial.

RQ3 *How does the runtime of #SAT solvers correlate to structural metrics of the feature model?* The runtime required to compute the cardinality of a feature model generally increases if the feature model grows in size or complexity. There is a strong or very strong positive correlation between the runtime of #SAT solvers and several structural metrics related to size and complexity, namely number of features, number of leaf features, number of constraints, number of clauses, and number of literals.

Feature models with few features or constraints seem to be simple to analyze for #SAT solvers. Each subject system with less than 1,000 features was evaluated within one second by at least one solver, independent of the number of constraints. Analogously, all subject systems with less than 1,000 constraints were evaluated by at least one solver within one second, independent of the number of features.

While both systems for which no solver computed a result have at least 3,500 constraints, a large number of features, or constraints do not necessarily cause a time-consuming computation. The *Automotive02* system contains by far the most features

(18,616), but `sharpSAT` still evaluated it in less than a second. The reason probably lies in the comparatively low number of constraints (1,369) while *Linux* and *Automotive05* contain 3,545 and 10,321 constraints, respectively. Furthermore, *uClinux-base* contains 3,455 constraints but the fastest solver is about 50 times faster than for *Automotive02* which contains only 1,369 constraints.

Our insights indicate that, independent of structural metrics, one of the fastest solvers should be used. There is no single feature model for which one of the fastest eight solvers fails, while another #SAT solver computes a result. Thus, we expect that if the fastest solvers do not scale to a feature model, the others will also fail.

Predicting performance based on structural metrics may still be beneficial. For instance, `countAntom` is slower than `sharpSAT` for 12 out 13 (successfully evaluated) subject systems, but significantly faster for all 116 CDL feature models. Applying a meta #SAT solver that selects a suitable #SAT solver for a given feature model should yield runtime benefits. Our insights on the correlations between structural metrics and runtime may be a useful starting point for future work on predicting performance. In particular, the metrics showing a strong correlation are promising indicators for predicting performance.

RQ4 *How do approximate #SAT solvers perform on industrial feature models?* Approximating the results with the evaluated approximate #SAT solvers yields no benefits as we can acquire exact results with shorter runtimes. In particular, the fastest exact solver `sharpSAT` is significantly faster than both approximate #SAT solvers for every single successfully evaluated feature model. The slower solver `ApproxMC` computed a result only for the two smallest considered feature models. While `ApproxCount` computed a result for the majority of models, it scaled to fewer feature models than the fastest exact #SAT solver `sharpSAT`.

A reason for the worse performance of approximate #SAT solvers may be that the solvers were evaluated on (and eventually optimized for) formulas from different domains with generally fewer satisfying assignments. The largest formulas evaluated induce up to 10^{12} (Chakraborty et al. 2013) and up to 10^{33} (Wei and Selman 2005) satisfying assignments, respectively (compared to up to 10^{1534} in our evaluation). Optimizing those approximations for formulas representing feature models may be beneficial.

8 Threats to validity

We identified the following potential threats to validity for our evaluation.

Translating the subject systems to feature models It is possible that the translation from the original proprietary format into a feature model changes the variability. Knüppel et al. remarked some threats to internal validity regarding their translation of product lines (Knüppel et al. 2017). First, there are differences between feature model semantics and the semantics of the variability languages used for CDL and KConfig. Second, the translation may have removed a few cross-tree constraints. Third, a few cases lead to features that did not appear in the input format (Knüppel et al. 2017). Still, this is the largest available benchmark and has been used by other authors (Krieter et al. 2018; Plazar et al. 2019; Baranov et al. 2020). Pett et al. (2021) translated the BusyBox model to CNF using KClause (Oh et al. 2019).[19] Then, the authors translated the CNF into feature model that is equivalent to

[19] https://doi.org/10.5281/zenodo.2574218

the CNF and, thus, should maintain the variability. In addition to publicly available subject systems, we translated three automotive product lines into feature models from a proprietary format. It is possible that we misinterpreted given constraints. However, we created the parser in direct cooperation with company interns. Furthermore, the interns reviewed the resulting feature models.

Translating feature models to the DIMACS format An incorrect translation of feature models to CNF may lead to incorrect cardinalities. Another important aspect of the translation to CNF is that the number of satisfying assignments has to be equal for the resulting CNF. This is not given for every conversion method (Knüppel et al. 2017). For every translation to CNF in DIMACS format, we used the FeatureIDE library (Kästner et al. 2009). FeatureIDE uses a transformation that does not introduce new variables nor changes the number of solutions. Nevertheless, we performed the following sanity checks to ensure a correct translation. First, we manually computed the model count of small feature models (¡ 100 valid configurations and only few cross-tree constraints) and compared these results with the ones computed by the solvers. Second, we made changes to the feature model that should change the model count in a certain way. For example, we added an optional feature to the root which should always double the number of valid configurations and verified that the #SAT solvers computed the expected result.

We did not consider the time required to translate the feature model to CNF. However, the translation time is equivalent for all #SAT solvers as they use the same CNF. Furthermore, for all feature models but Linux the translation required only a few milliseconds.

Wrapper for BuDDy and Cudd As described in Section 5.2, we used a wrapper to interface with `BuDDy` and `Cudd`, due to their incapability to process DIMACS directly. The implementation of our wrapper for `BuDDy` and `Cudd` could be erroneous or inefficient, yielding a negative impact on their performance. While parsing of the input is handled by the wrapper, the BDDs are constructed entirely by `BuDDy` and `Cudd` using the parameters and techniques suggested in their respective manuals. For each successful computation of `BuDDy` and `Cudd`, the returned number of satisfying assignments was correct. Furthermore, for every feature model the parsing time of the wrapper required less than one second and at most 10% of the overall runtime. Note that each time `BuDDy` or `Cudd` required more than one second of runtime the relative share of parsing time is even lower (at most 2%). Thus, we consider the parsing time of the DIMACS input is negligible compared to the overall runtime and do not expect an impact on our conclusions on the performance of `BuDDy` or `Cudd`. Furthermore, we decided against using the widely used (Mendonça 2009; Mendonca and Cowan 2010; Pohl et al. 2011) library JavaBDD as it misses support for the latest version of `Cudd` (3.0.0) and frequently crashes when using `BuDDy`.

Parameterization of the solvers Typically, there are various parameters to adapt the behavior of the solvers, such as enabling or disabling boolean constraint propagation. These parameters may have a noticeable impact on the scalability in some cases (Wei and Selman 2005). In general, we used the default parameterization for each solver to achieve the following: (1) prevent introducing a bias based on our decision of the parameterization, and (2) evaluate the solver's performance when integrated without further expertise which we typically expect in practice. In general, evaluating multiple parameter permutations multiplicatively vastly increases the complexity, required time, and ecological footprint of the performed experiments.

Correctness of the solvers We used only external solvers without a possibility of directly verifying the results. However, for every subject system the number of satisfying assignments returned by each solver was equal. This is a strong indicator for the correctness of the solvers. Furthermore, we manually computed the cardinality for multiple small feature models (¡ 100 valid configurations) and compared them to the results of #SAT solvers.

Computational bias When performing measurements, it is possible that a program accelerates during the computations. In this case, early measurements might be slower than later ones. In our benchmark framework, each single invocation of a #SAT solver is performed in a separate execution of the binary. Thus, the solvers are in the same state at the start of each computation. It may be possible that hardware optimizations induce a warm-up. We disabled *turboboost* and *file system cache* to reduce a potential bias.

In general, it is possible for a background process to influence the runtime of a solver and, thus, impact our results. First, we disabled hyper-threading. Second, we performed a preliminary experiment with five repetitions for each solver several months prior to the evaluation described in this work which resulted in the same conclusions regarding the performance of the solvers. Third, during the 50 repetitions of Experiment 1b, the solvers always had very similar runtimes for the same feature model. Fourth, during the runtime of the experiments no other computational expensive task was performed on the device and each measurement was performed sequentially. Fifth, we occasionally monitored the available RAM and CPU resources. Every time we tracked, there were at least 40 GB of RAM available and less than 15% of the CPU used. Therefore, we do not expect that any conclusion we made is impacted by a background process.

Single measurements for slow solvers For slow solvers, we only performed one repetition per measurement. It is possible that for 50 repetitions the median significantly differs from a single measurement for some feature models. Nevertheless, neither solver was excluded after Experiment 1a due to single or few measurements but due to a large gap to the fastest #SAT solvers.

Random effects It is possible that the runtimes of #SAT solvers are affected by random effects. For example, `c2d` (Darwiche 2002, 2004) randomly chooses cuts in order to create a decomposition tree of the formula at the start of the computation. To reduce the bias resulting from randomness, we performed 50 repetitions in Experiment 1b and Experiment 2b and performed statistical tests on the significance of results.

External validity solvers Our results cannot be necessarily transferred to other #SAT solvers. For instance, Kübler etal (Kübler et al. 2010) developed their own tool to compute the cardinality of feature models. Their tool is not publicly available and, thus, we could not evaluate and compare it to other solvers. Nevertheless, we evaluated a large variety of different #SAT solvers. To the best of our knowledge, we included each publicly available #SAT solver in our benchmark.

External validity systems We cannot claim that our results can be transferred to any other industrial product lines. However, we considered multiple domains, namely automotive, operating system, database, and financial services to increase our confidence. We overall evaluated 130 feature models which cover a wide range of number of features (76–18,616), number of constraints (20–10,321), number of valid configurations ($\approx 10^9$–10^{1534}), and runtime of #SAT solvers (between few milliseconds and hitting a timeout of 24 hours).

Therefore, we expect that our results represent a reasonable indicator for the scalability of #SAT solvers on other product lines.

9 Related work

In this section, we discuss work that is related to ours regarding (1) applying #SAT to feature models, (2) usage of #SAT technology in feature-modelling tools, and (3) computing the cardinality with tools that are not based on propositional logic.

Applying #SAT to feature models Kübler et al. (2010) also evaluated the use of two #SAT solvers, `Cachet` (Sang et al. 2004) and `c2d` (Darwiche 2004), on three different versions of an automotive product line. We evaluated both solvers and they were outperformed by newer solvers on most instances. However, the authors also proposed their own model counter that was not based on conjunctive normal form and performed better than `Cachet` and `c2d`. However, their solver and their evaluated product lines are not publicly available. Therefore, we could not directly compare the results. Overall, we evaluated 21 solvers on 130 formulas while Kübler et al. evaluated 3 solvers on 3 formulas.

Pohl et al. (2011) evaluated different feature model analyses including model counting using BDDs, constraint-satisfaction-problem solvers, and SAT solvers. However, the authors used models with much smaller configuration spaces and fewer features for their evaluation. Their analyzed configuration spaces only reached up to 10^8 valid configurations whereas 97.3% of our feature models have larger configuration spaces with up-to 10^{1534} valid configurations.

Oh et al. (2019) evaluated the application of #SAT for uniform random sampling with their tool *Smarch*. Their results indicate that #SAT can be used to create a uniformly distributed sample for a variety of industrial feature models. However, their evaluation is limited to one application (uniform random sampling) and limited to one solver (`sharpSAT`). Sharma et al. (2018) proposed using #SAT technology for uniform random sampling and provided an algorithm exploiting d-DNNFs. However, their empirical evaluation is also limited to uniform random sampling and two solvers (`d4` and `dSharp`). We evaluate 21 solvers including the three solvers considered by Oh et al. (2019) and Sharma et al. (2018).

Current tool support for #SAT technology BDDs are a popular choice for counting the number of valid configurations in a product line as it is possible to compute the BDD offline and then compute the cardinality with linear time in the number of nodes (Acher et al. 2013; Hadzic et al. 2004; Mendonça et al. 2009). However, our results indicate that existing BDD libraries do not scale to industrial feature models. Additionally, d-DNNFs can be computed offline as well and performed significantly better than BDDs in all our experiments (Darwiche and Marquis 2002).

FeatureIDE uses a regular SAT solver (`SAT4J` (Le Berre and Parrain 2010)) to compute the number of valid configurations (Thüm et al. 2011). The tool realizes counting with a regular SAT solver using blocking clauses (Toda and Soh 2016); after finding a valid assignment α, the negation of α is added as a clause to the formula. Thus, α is not a valid assignment for the resulting formula and the next run of the solver returns another assignment until no new satisfying assignments are left. For each satisfying assignment (i.e., valid configuration), an invocation of the SAT solver is required. Our results indicate that

industrial feature models induce up to 10^{1500} valid configurations. Therefore, the algorithm should not scale for larger systems.

Non-propositional model counting Constraint satisfaction problems (CSP) are an alternative to propositional logic for the representation of feature models (Benavides et al. 2005, 2006, 2010; Pohl et al. 2011). CSPs are defined by a set of variables, domains for each variable, and constraints over these variables. For CSPs, the variables may also be integers or intervals, contrary to propositional boolean variables which are strictly binary (Benavides et al. 2010). Benavides et al. (2005) use constraint programming (CP) to compute the number of valid configurations for feature models. However, the models considered in their experiment only included up to 23 features (Benavides et al. 2005). Pohl et al. (2011) compare SAT solvers, BDDs, and CSP solvers for several feature-model analyses that include computing the cardinality. Their results indicate that the analyzed CSP solvers scale far worse than the #SAT solvers evaluated in our experiment (Pohl et al. 2011). Munoz et al. (2019) examined counting the number of valid configurations of feature models with numerical features for uniform random sampling. The authors evaluated an SMT solver, a CP solver, and the #SAT solver `sharpSAT`. The numerical values were translated to propositional logic using bit-blasting (Munoz et al. 2019). In their experiment, `sharpSAT` outperformed the CP and SMT solver. This indicates that #SAT solvers are also a reasonable choice for computing the number of valid configurations for feature models with numerical values and our results (e.g., recommendations of solvers) could also be useful for non-propositional model counting.

10 Future work

In this section, we describe further tasks in applying #SAT solvers to industrial feature models.

Cardinality of features and partial configurations In this work, we limited our empirical evaluation to computing the cardinality of feature models (i.e., the number of valid configurations of the entire feature model). In our previous work (Sundermann et al. 2021), we presented 21 applications and a major part of them is dependent on the cardinality of (possibly many) features (i.e., number of valid configurations that contain a specific feature) or the cardinality of partial configurations (i.e., number of valid configurations that include some and exclude some other features). The runtimes of computing the cardinality of the entire feature model (as measured in our empirical evaluation) can be used as estimate for computing the cardinality of a feature or a partial configuration due to the similar input formulas (Sundermann et al. 2021). Nevertheless, to provide accurate insights on the scalability of these applications, an empirical evaluation for computing the cardinality of features and partial configurations is required.

Analyzing #SAT during the evolution of systems Often, product lines evolve over time (Svahnberg and Bosch 1999). Typically, underlying feature models grow both in number of features and constraints (Sundermann et al. 2020; Israeli and Feitelson 2010). As we found a strong correlation between the scalability of #SAT and both metrics (i.e., number of features and number of constraints), the evolution of a system may increase the runtime required to evaluate an underlying feature model with a #SAT solver. This is also indicated

by the preliminary results of our previous work (Sundermann et al. 2020). If a product lines evolves over time, even product lines for which #SAT solvers scale currently may be infeasible to analyze in the future or vice versa.

Exploit d-DNNFs for cardinality-based analyses In our empirical evaluation, all three d-DNNF compilers, namely `dSharp`, `d4`, and `c2d` were part of the eight fastest solvers. If we require multiple computations on a single feature model (e.g., to compute the cardinalities for multiple features or partial configurations), exploiting a compiled d-DNNF may be beneficial. However, the research on exploiting an existing d-DNNF is very limited (Sharma et al. 2018) as most work focuses on the compilation process (Darwiche 2002, 2004; Lagniez and Marquis 2017; Oztok and Darwiche 2014; Muise et al. 2010; Huang and Darwiche 2005). While SDDs and BDDs are also considerable target formats for knowledge compilation, all compilers based on these formats performed significantly worse than `dSharp` and `d4`.

Parameterize #SAT solvers In this paper, we invoked the #SAT solvers using the default parameters with a few exceptions (e.g., some solvers require specific parameters to perform #SAT instead of SAT). Other parameterizations (e.g., selecting strategies for variable ordering) may improve the performance of #SAT solvers. Especially, the runtime of approximate #SAT solvers is dependent on the given parameters. However, identifying effective parameters is not trivial. To use #SAT solvers to their full potential requires finding suitable parameters that result in efficient and effective computations.

Further metrics for a meta-solver Our results show that the solvers perform differently depending on the system. None of the solvers is faster than all other solvers for every feature model. Analyzing structural metrics of the feature model may enable an efficient meta-solver that selects the most promising solver depending on a given instance. For regular SAT, it is already known that selecting a solver based on a given formula often improves the performance (Xu et al. 2008).

Directly translate feature models to target format For every experiment, we used propositional formulas in conjunctive normal form. The translation to CNF was not considered in the runtime. However, for the larger systems, the translation requires a considerable amount of time. Directly translating the feature model to knowledge compilation target formats, such as BDDs or d-DNNFs, might result in two benefits. First, the time overhead of translating the model to CNF would be eliminated. Second, using structural information of the feature model may accelerate the translation to the target format.

Purpose-built solvers for analyzing feature models None of the analyzed #SAT solvers and knowledge compilers is optimized for feature models. Optimizing the computations specifically for feature models may improve the performance of solvers. One improvement may be deriving beneficial variable orders using structural information of the feature model. The performance of each considered type of solver is highly dependent on variable ordering

(Sang et al. 2005; Muise et al. 2010; Thurley 2006; Wei and Selman 2005; Darwiche 2002; Toda and Soh 2016).

11 Conclusion

A large variety of feature-model analyses is dependent on computing the cardinality of features models (Sundermann et al. 2021). However, the scalability of such analyses is largely unknown. We analyzed 19 exact and 2 approximate #SAT solvers on the task of computing the cardinality of industrial feature models. Overall, we evaluated the #SAT solvers on 130 feature models from 15 subject systems.

Our results strongly indicate that current #SAT solvers scale to many, but not to all systems. Out of the 15 evaluated systems, eight solvers computed the cardinality of 13 (86.7%) systems within 10 minutes per system. The solver with the overall shortest runtime is `sharpSAT` requiring less than three seconds for all 13 models in total. However, for the two remaining systems, namely *Linux* and *Automotive05*, none of the solvers was able to compute a result within 24 hours of runtime.

While no solver was strictly superior to all other solvers, we identified several promising #SAT solvers for the task of computing the cardinality of feature models. For single #SAT computations on feature models, we recommend using the DPLL-based solvers `sharpSAT`, `countAntom`, and `Ganak`. For applications requiring multiple #SAT invocations, reusing d-DNNFs seems promising. All three considered d-DNNF compilers, namely `dSharp`, `d4`, and `c2d`, were within the fastest eight solvers. Surprisingly, each approximate #SAT solver we evaluated is significantly slower than the fastest exact #SAT solver for every considered feature model and, thus, yields no benefits over the exact solvers.

The runtime of all #SAT solvers tends to increase for feature models with a larger number of constraints or features. Each feature model with either fewer than 1,000 features or fewer than 1,000 constraints was evaluated within one second by the solver with the shortest runtime for that feature model. Nevertheless, the results indicate that a higher number of constraints or features does not necessarily result in longer runtimes.

Acknowledgments We sincerely thank the reviewers and editors for their valuable feedback on our work. Further, we thank Heiko Raab and Rahel Arens for their help in improving the manuscript. This work has been partially supported by the German Research Foundation within the project VariantSync (TH 2387/1-1).

Funding Open Access funding enabled and organized by Projekt DEAL.

Data Availability The datasets generated during and/or analyzed during the current study are available in the replication repository, https://doi.org/10.5281/zenodo.7329979.

Declarations

Conflict of Interests The authors have no conflicts of interest to declare that are relevant to the content of this article.

Open Access This article is licensed under a Creative Commons Attribution 4.0 International License, which permits use, sharing, adaptation, distribution and reproduction in any medium or format, as long as you give appropriate credit to the original author(s) and the source, provide a link to the Creative Commons licence, and indicate if changes were made. The images or other third party material in this article are included in the article's Creative Commons licence, unless indicated otherwise in a credit line to the material. If material is not included in the article's Creative Commons licence and your intended use is not permitted by statutory regulation or exceeds the permitted use, you will need to obtain permission directly from the copyright holder. To view a copy of this licence, visit http://creativecommons.org/licenses/by/4.0/.

References

Acher M, Collet P, Lahire P, France RB (2013) Familiar: a domain-specific language for large scale management of feature models. Sci Comput Program (SCP) 78(6):657–681

Ananieva S, Kowal M, Thüm T, Schaefer I (2016) Implicit constraints in partial feature models. In: Proc. int'l workshop on feature-oriented software development (FOSD), ACM, pp 18–27

Ansótegui C, Bonet ML, Levy J (2009) On the structure of industrial SAT instances. In: Proc. int'l conf.on principles and practice of constraint programming (CP), Springer, pp 127–141

Apel S, Batory D, Kästner C, Saake G (2013) Feature-Oriented Software Product Lines. Springer

Artusi R, Verderio P, Marubini E (2002) Bravais-Pearson and spearman correlation coefficients: meaning, test of hypothesis and confidence interval. Int J Biol Markers (IJBM) 17(2):148–151

Bagheri E, Gasevic D (2011) Assessing the maintainability of software product line feature models using structural metrics. Softw Qual J (SQJ) 19(3):579–612

Bagheri E, Noia TD, Gasevic D, Ragone A (2012) Formalizing interactive staged feature model configuration. J Softw Evol Process 24(4):375–400

Baranov E, Legay A, Meel KS (2020) Baital: an adaptive weighted sampling approach for improved t-wise coverage. In: Proc. Europ. Software engineering conf./foundations of software engineering (ESEC/FSE), ACM, pp 1114–1126

Batory D (2005) Feature models, grammars, and propositional formulas. In: Proc. int'l systems and software product line conf. (SPLC), Springer, pp 7–20

Bayardo Jr RJ, Pehoushek JD (2000) Counting models using connected components. In: Proc. Conf. on artificial intelligence (AAAI), AAAI press, pp 157–162

Benavides D, Segura S, Ruiz-Cortés A (2010) Automated analysis of feature models 20 years later a literature review. Inf Syst 35(6):615–708

Benavides D, Segura S, Trinidad P, Ruiz-Cortés A (2006) Using java CSP solvers in the automated analyses of feature models. In: Proc. generative and transformational techniques in software engineering, Springer, pp 399–408

Benavides D, Segura S, Trinidad P, Ruiz-Cortés A (2007) FAMA: Tooling a framework for the automated analysis of feature models. In: Proc. int'l workshop on variability modelling of software-intensive systems (VaMoS), lero, pp 129–134. Technical report 2007-01

Benavides D, Trinidad P, Ruiz-Cortés A (2005) Using constraint programming to reason on feature models. In: Proc. Int'l conf. on software engineering and knowledge engineering (SEKE), pp 677–682

Bezerra CIM, Andrade RMC, Monteiro JMS (2014) Measures for quality evaluation of feature models. In: Proc. Int'l conf. on software reuse (ICSR), Springer, pp 282–297

Biere A (2008) PicoSAT Essentials. J. Satisfiability Boolean Model Comput 4:75–97

Biere A, Heule M, van Maaren H, Walsh T (2009) Handbook of satisfiability: volume 185 frontiers in artificial intelligence and applications. IOS press

Bosch J, Florijn G, Greefhorst D, Kuusela J, Obbink JH, Pohl K (2001) Variability issues in software product lines. In: Proc. int'l workshop on software product-family engineering (PFE), Springer, pp 13–21

Bryant RE (1986) Graph-Based Algorithms for boolean function manipulation. IEEE Trans Comput C-35(8):677–691

Bryant RE (2018) Binary decision diagrams. In: Handbook of model checking, Springer, pp 191–217

Burchard J, Schubert T, Becker B (2015) Laissez-Faire caching for parallel# SAT solving. In: Proc. Int'l conf. on theory and applications of satisfiability testing (SAT), Springer, pp 46–61

Chakraborty S, Meel KS, Vardi MY (2013) A scalable approximate model counter. In: Schulte C (ed) Proc. Int'l Conf. on Principles and Practice of Constraint Programming (CP), Springer, pp 200–216

Chen S, Erwig M (2011) Optimizing the product derivation process. In: Proc. Int'l systems and software product line conf. (SPLC), IEEE, pp 35–44

Conover WJ, Iman RL (1981) Rank transformations as a bridge between parametric and nonparametric statistics. Am Stat 35(3):124–129

Czarnecki K, Wàsowski A (2007) Feature diagrams and logics: There and back again. In: Proc. Int'l systems and software product line conf. (SPLC), IEEE, pp 23–34

Darwiche A (2001) On the tractable counting of theory models and its application to truth maintenance and belief revision. J Appl Non-Class Logics 11(1-2):11–34

Darwiche A (2002) A compiler for deterministic, decomposable negation normal form. In: Proc. Conf. on artificial intelligence (AAAI), AAAI press, pp 627–634

Darwiche A (2004) New advances in compiling CNF to decomposable negation normal form. In: Proc. Europ. Conf. on artificial intelligence, IOS press, pp 318–322

Darwiche A (2011) SDD: a new canonical representation of propositional knowledge bases. AAAI Press, pp 819–826

Darwiche A, Marquis P (2002) A knowledge compilation map. J Artif Intell Res (JAIR) 17(1):229–264
Fernández-Amorós D, Heradio R, Cerrada JA, Cerrada C (2014) A scalable approach to exact model and commonality counting for extended feature models. IEEE Trans Softw Eng (TSE) 40(9):895–910
Fichte JK, Hecher M, Hamiti F (2021) The model counting competition 2020. ACM J Exp Algorithmics (JEA) 26:1–26 , 13:1–26
Fichte JK, Hecher M, Zisser M (2019) An Improved GPU-based SAT Model Counter. In: Proc. Int'l conf. on principles and practice of constraint programming (CP), Springer, pp 491–509
Fritsch C, Abt R, Renz B (2020) The benefits of a feature model in banking. In: Proc. Int'l systems and software product line conf. (SPLC). ACM
Galindo JA, Acher M, Tirado JM, Vidal C, Baudry B, Benavides D (2016) Exploiting the enumeration of all feature model configurations a new perspective with distributed computing. In: Proc. Int'l systems and software product line conf. (SPLC), ACM, pp 74–78
Gomes CP, Sabharwal A, Selman B (2006) Model counting: a new strategy for obtaining good bounds. In: Proc. Conf. on artificial intelligence (AAAI), AAAI press, pp 54–61
Hadzic T, Subbarayan S, Jensen RM, Andersen HR, Møller J, Hulgaard H (2004) Fast backtrack-free product configuration using a precompiled solution space representation. In: Proc. Int'l conf. on economic, technical and organisational aspects of product configuration systems, Gamez publishing, pp 131–138
Heradio R, Fernández-Amorós D, Cerrada JA, Abad I (2013) A literature review on feature diagram product counting and its usage in software product line economic models. Int J Softw Eng Knowl Eng (IJSEKE) 23(08):1177–1204
Heradio R, Fernández-Amorós D, Mayr-Dorn C, Egyed A (2019) Supporting the statistical analysis of variability models. In: Proc. Int'l conf. on software engineering (ICSE), IEEE, pp 843–853
Heradio R, Pérez-Morago HJ, Fernández-Amorós D, Bean R, Cabrerizo FJ, Cerrada C, Herrera-Viedma E (2016) Binary decision diagram algorithms to perform hard analysis operations on variability models. In: Proc. Int'l Conf. on Intelligent Software Methodologies, Tools and Techniques (SOMET), IOS Press, pp 139–154
Heradio-Gil R, Fernández-Amorós D, Cerrada JA, Cerrada C (2011) Supporting commonality-based analysis of software product lines. IET Softw 5(6):496–509
Heß T, Sundermann C, Thüm T (2021) On the scalability of building binary decision diagrams for current feature models. In: Proc. Int'l systems and software product line conf. (SPLC), ACM, pp 131–135
Huang J, Darwiche A (2005) DPLL with a trace: from SAT to knowledge compilation. In: Proc. Int'l joint conf. on artificial intelligence (IJCAI), vol 5. Professional book center, pp 156–162
Huth M, Ryan M (2004) Logic in computer science: modelling and reasoning about systems. Cambridge University Press
Israeli A, Feitelson DG (2010) The linux kernel as a case study in software evolution. J. Syst Softw (JSS) 83(3):485–501
Kästner C, Apel S, Batory D (2007) A case study implementing features using aspectJ. In: Proc. Int'l systems and software product line conf. (SPLC), IEEE, pp 223–232
Kästner C, Thüm T, Saake G, Feigenspan J, Leich T, Wielgorz F (2009) FeatureIDE: a tool framework for feature-oriented software development. In: Proc. Int'l Conf. on Software Engineering (ICSE), IEEE, pp 611–614. Formal demonstration paper
Klebanov V, Manthey N, Muise C (2013) SAT-based analysis and quantification of information flow in programs. In: Proc. Int'l conf. on quantitative evaluation of systems (QEST), Springer, pp 177–192
Knüppel A, Thüm T, Mennicke S, Meinicke J, Schaefer I (2017) Is there a mismatch between Real-World feature models and Product-Line research? In: Proc. Europ. software engineering conf. Foundations of software engineering (ESEC/FSE), ACM, pp 291–302
Koriche F, Lagniez J-M, Marquis P, Thomas S (2013) Knowledge compilation for model counting affine decision trees. In: Proc. Int'l joint conf. on artificial intelligence (IJCAI), AAAI press
Kowal M, Ananieva S, Thüm T (2016) Explaining anomalies in feature models. In: Proc. Int'l conf. on generative programming: concepts & experiences (GPCE), ACM, pp 132–143
Krieter S, Pinnecke M, Krüger J, Sprey J, Sontag C, Thüm T, Leich T, Saake G (2017) Feature IDE empowering third-party developers. In: Proc. Int'l systems and software product line conf. (SPLC), ACM, pp 42–45
Krieter S, Thüm T, Schulze S, Schröter R, Saake G (2018) Propagating configuration decisions with modal implication graphs. In: Proc. Int'l conf. on software engineering (ICSE), ACM, pp 898–909
Kübler A, Zengler C, Küchlin W (2010) Model counting in product configuration. In: Proc. Int'l workshop on logics for component configuration (lococo), Open publishing association, pp 44–53
Lagniez J-M, Marquis P (2017) An improved decision-DNNF compiler. In: Proc. Int'l joint conf. on artificial intelligence (IJCAI), pages 667–673. International joint conferences on artificial intelligence

Lagniez J-M, Marquis P, Szczepanski N (2018) DMC: a distributed model counter. In: Proc. Int'l joint conf. on artificial intelligence (IJCAI), pp 1331–1338. International joint conferences on artificial intelligence
Le Berre D, Parrain A (2010) The sat4j library, release 2.2. J Satisfiability Boolean Model Comput 7(2-3):59–64
McKnight PE, Najab J (2010) Mann-Whitney U test. Corsini Encycl Psychol 1–1
Mendonca M, Cowan D (2010) Decision-making coordination and efficient reasoning techniques for Feature-Based configuration. Sci Comput Program (SCP) 75(5):311–332
Mendonça M (2009) Efficient reasoning techniques for large scale feature models. University of Waterloo, PhD thesis
Mendonça M, Branco M, Cowan D (2009) S.P.L.O.T.: software product lines online tools. In: Proc. Conf. on object-oriented programming, systems, languages and applications (OOPSLA), ACM, pp 761–762
Mendonça M, Wàsowski A, Czarnecki K (2009) SAT-Based analysis of feature models is easy. In: Proc. Int'l systems and software product line conf. (SPLC), Software engineering institute, pp 231–240
Muise C, McIlraith S, Beck JC, Hsu E (2010) Fast d-DNNF Compilation with sharpSAT. In: Proc. Conf. on artificial intelligence (AAAI). AAAI press
Munoz D-J, Oh J, Pinto M, Fuentes L, Batory D (2019) Uniform random sampling product configurations of feature models that have numerical features. In: Proc. Int'l systems and software product line conf. (SPLC), ACM, pp 289–301
Nieke M, Mauro J, Seidl C, Thüm T, Yu IC, Franzke F (2018) Anomaly analyses for Feature-Model evolution. In: Proc. Int'l conf. on generative programming: Concepts & experiences (GPCE), ACM, pp 188–201
Oh J, Batory D, Myers M, Siegmund N (2016) Sampling: finding product line configurations with high performance by random technical report University of Texas at Austin
Oh J, Batory D, Myers M, Siegmund N (2017) Finding Near-Optimal configurations in product lines by random sampling. In: Proc. Int'l symposium on foundations of software engineering (FSE), pp 61–71
Oh J, Gazzillo P, Batory D, Heule M, Myers M (2019) Uniform Sampling from Kconfig Feature Models. Technical Report TR-19-02, The university of texas at austin department of computer science
Oztok U, Darwiche A (2014) On compiling CNF into decision-DNNF. In: Proc. Int'l conf. on principles and practice of constraint programming (CP), Springer, pp 42–57
Oztok U, Darwiche A (2015) A Top-Down compiler for sentential decision diagrams. In: Proc. Int'l joint conf. on artificial intelligence (IJCAI), AAAI press, pp 3141–3148
Pérez Morago HJ (2016) BDD Algorithms to perform hard analysis operations on variability models. Universidad Nacional de Educación a Distancia, PhD thesis
Perrouin G, Sen S, Klein J, Baudry B, Le Traon Y (2010) Automated and scalable T-Wise test case generation strategies for software product lines. In: Proc. Int'l conf. on software testing, verification and validation (ICST), IEEE, pp 459–468
Pett T, Krieter S, Runge T, Thüm T, Lochau M, Schaefer I (2021) Stability of Product-Line sampling in continuous integration. In: Proc. Int'l working conf. on variability modelling of software-intensive systems (VaMoS). ACM
Pett T, Thüm T, Runge T, Krieter S, Lochau M, Schaefer I (2019) Product sampling for product lines the scalability challenge. In: Proc. Int'l systems and software product line conf. (SPLC), ACM, pp 78–83
Plazar Q, Acher M, Perrouin G, Devroey X, Cordy M (2019) Uniform sampling of SAT solutions for configurable systems are we there yet? In: Proc. Int'l conf. on software testing, verification and validation (ICST), IEEE, pp 240–251
Pohl R, Lauenroth K, Pohl K (2011) A performance comparison of contemporary algorithmic approaches for automated analysis operations on feature models. In: Proc. Int'l conf. on automated software engineering (ASE), IEEE, pp 313–322
Rudell R (1993) Dynamic variable ordering for ordered binary decision diagrams. In: Proc. Int'l conf. on computer-aided design (ICCAD), IEEE, pp 42–47
Sang T, Bacchus F, Beame P, Kautz HA, Pitassi T (2004) Combining component caching and clause learning for effective model counting. In: Proc. Int'l conf. on theory and applications of satisfiability testing (SAT), Springer, pp 20–28
Sang T, Beame P, Kautz H (2005) Heuristics for fast exact model counting. In: Proc. Int'l conf. on theory and applications of satisfiability testing (SAT), Springer, pp 226–240
Schröter R, Krieter S, Thüm T, Benduhn F, Saake G (2016) Feature-model interfaces: the highway to compositional analyses of Highly-Configurable systems. In: Proc. Int'l conf. on software engineering (ICSE), ACM, pp 667–678
Segura S (2008) Automated analysis of feature models using atomic sets. In: Proc. Int'l systems and software product line conf. (SPLC), vol 2. IEEE, pp 201–207

Sharma S, Gupta R, Roy S, Meel KS (2018) Knowledge compilation meets uniform sampling. In: Proc. Int'l conf. on logic for programming, artificial intelligence, and reasoning, Easy chair, pp 620–636
Sharma S, Roy S, Soos M, Meel KS (2019) GANAK: a scalable probabilistic exact model counter. In: Proc. Int'l joint conf. on artificial intelligence (IJCAI), vol 19. AAAI press, pp 1169–1176
Sobernig S, Apel S, Kolesnikov S, Siegmund N (2016) Quantifying structural attributes of system decompositions in 28 Feature-Oriented software product lines. Empir Softw Eng (EMSE) 21(4):1670–1705
Sprey J, Sundermann C, Krieter S, Nieke M, Mauro J, Thüm T, Schaefer I (2020) SMT-based variability analyses in featureIDE. In: Proc. Int'l working conf. on variability modelling of software-intensive systems (VaMoS). ACM
Sullivan GM, Feinn R (2012) Using effect size—or why the P value is not enough. J Grad Med Educ 4(3):279–282
Sundermann C, Nieke M, Bittner PM, Heß T, Thüm T, Schaefer I (2021) Applications of #SAT solvers on feature models. In: Proc. Int'l working conf. on variability modelling of software-intensive systems (VaMoS). ACM
Sundermann C, Thüm T, Schaefer I (2020) Evaluating #SAT solvers on industrial feature models. In: Proc. Int'l working conf. on variability modelling of software-intensive systems (VaMoS). ACM
Svahnberg M, Bosch J (1999) Evolution in software product lines two cases. J Softw Maint (JSM) 11(6):391–422
Thüm T, Kästner C, Erdweg S, Siegmund N (2011) Abstract features in feature modeling. In: Proc. Int'l systems and software product line conf. (SPLC), IEEE, pp 191–200
Thurley M (2006) SharpSAT - counting models with advanced component caching and implicit BCP. In: Proc. Int'l conf. on theory and applications of satisfiability testing (SAT), Springer pp 424–429
Toda T, Soh T (2016) Implementing efficient all solutions SAT solvers. ACM J Exp Algorithmics (JEA) 21(1):1.12:1–1.12:44
Valiant LG (1979) The complexity of computing the permanent. Theor Comput Sci 8(2):189–201
Wei W, Selman B (2005) A new approach to model counting. In: Proc. Int'l conf. on theory and applications of satisfiability testing (SAT), Springer, pp 324–339
Xu L, Hutter F, Hoos HH, Leyton-Brown K (2008) SATzilla: portfolio-based algorithm selection for SAT. J Artif Intell Res (JAIR) 32:565–606
Zar JH (1972) Significance testing of the spearman rank correlation coefficient. J Am Stat Assoc 67(339):578–580

Publisher's note Springer Nature remains neutral with regard to jurisdictional claims in published maps and institutional affiliations.

Affiliations

Chico Sundermann[1] · Tobias Heß[1] · Michael Nieke[2] · Paul Maximilian Bittner[1] · Jeffrey M. Young[3] · Thomas Thüm[1] · Ina Schaefer[2]

Tobias Heß
tobias.hess@uni-ulm.de

Michael Nieke
m.nieke@tu-bs.de

Paul Maximilian Bittner
paul.bittner@uni-ulm.de

Jeffrey M. Young
jeffrey.young@iohk.io

Thomas Thüm
thomas.thuem@uni-ulm.de

Ina Schaefer
i.schaefer@tu-bs.de

1 University of Ulm, Ulm, Germany

2 Technische Universität Braunschweig, Brunswick, Germany

3 IOHK, Longmont, Colorado, United States

C.4 VaMoS'20: Evaluating #SAT Solvers on Industrial Feature Models

[B2] Chico Sundermann, Thomas Thüm, and Ina Schaefer. "Evaluating #SAT Solvers on Industrial Feature Models". In: *Proc. Int'l Working Conf. on Variability Modelling of Software-Intensive Systems (VaMoS)*. ACM, 2020. DOI: 10.1145/3377024.3377025

© 2020 is held by authors and they preserve rights to include a partial or complete version of the paper in a dissertation. Check https://doi.org/10.1145/3377024.3377025 for the published version.

Evaluating #SAT Solvers on Industrial Feature Models

Chico Sundermann
Technische Universität Braunschweig

Thomas Thüm
University of Ulm

Ina Schaefer
Technische Universität Braunschweig

ABSTRACT

Configurable systems are widely used for families of products that share multiple configuration options. These systems often induce a large configuration space. Handling the variability of such a system is difficult without being able to measure its complexity. Several methods depend on computing the number of valid configurations, such as estimating the effort of an update or effectively reducing the variability of a system. In many cases, it is possible to map a configurable system to propositional logic. Therefore, we use #SAT in order to evaluate variability of such systems. A #SAT solver computes the number of valid assignments of a propositional formula. However, this problem is even harder than SAT. The main contribution of our work is an investigation of the scalability of off-the-shelf #SAT solvers on industrial feature models. Additionally, we examine the correlation between size of a system and the runtime of a solver computing the number of valid configurations. In this paper, we empirically evaluate nine publicly available #SAT solvers on 127 industrial feature models. Our results indicate that current solvers master a majority of the evaluated systems. However, there are large models, for which none of the evaluated solvers scales. Nevertheless, there are even larger and more complex systems for which the solvers scale.

KEYWORDS

Configurable Systems, Feature Models, Product Lines, Model Counting, Configuration Counting, #SAT

1 INTRODUCTION

A configurable system represents a variety of valid configurations that share certain configuration options, also called features [7]. A configuration is induced by a selection of the features. However, systems typically contain constraints which limit the set of valid configurations. For example, a car product line can be interpreted as such a configurable system. A constraint to limit the set of valid configurations might be allowing only one gearbox type for each car: manual or automatic. Thus, a configuration with both or no type of gearbox is considered invalid.

The set of valid configurations is known as the configuration space [5]. When introducing a new feature to a configurable, system the number of valid configurations is doubled in the worst case. In this scenario, each previously valid configuration is still valid with and without the new feature. Thus, the configuration space grows up-to exponentially. A configurable system with n features has up-to 2^n valid configurations [9]. However, real-world systems are typically more limited [8, 27].

It is difficult to manage this variability without knowing the number of valid configurations induced by a configurable system. For instance, the following methods rely on counting the number of valid configurations: commonality and homogenity of features [9, 14, 17], variability reduction [9], rating errors in a configurable system [27], and uniform random sampling [33]. These use cases are further described in Section 2.

Model counting can be used for quantitative analysis of configurable systems [17, 27]. In this paper, we focus on propositional model counting (for short #SAT), which determines the number of valid assignments for a given propositional formula. #SAT is an even harder problem than SAT [12]. However, the #SAT community keeps pushing the limits of #SAT solvers [6, 12, 15, 28, 42]. While Mendonca et al. argued that SAT is typically easy for real-world models compared to hard instances of SAT [30], this is currently unknown for #SAT to the best of our knowledge.

The main goal of this paper is to examine the efficiency of current #SAT solvers on real-world configurable systems. In order to do so, we analyze nine publicly available solvers [6, 10, 12, 15, 28, 32, 39, 42, 43] on 127 industrial systems. Hereby, we focus on exact #SAT solvers, contrary to approximate solvers. The analyzed systems consist of a benchmark from Knüppel et al. [25] and three automotive product lines provided by our industry partner. In our evaluation, we aim to answer the following research questions:

- **RQ1:** Do #SAT solvers scale to industrial configuration spaces?
- **RQ2:** Is one #SAT solver superior to the other solvers?
- **RQ3:** Does the runtime of the solvers correlate to the size or complexity of the configuration space?
- **RQ4:** How does the number of valid configurations relate to the number of all configurations?
- **RQ5:** How does the number of valid configurations change during the evolution of a configurable system?

2 THE NEED FOR CONFIGURATION COUNTING

In this section, we describe use cases for counting the number of valid configurations of configurable systems.

Commonality. The relative share of valid configurations that contain a certain feature is called commonality of that feature [17]. It can be used as an indication for the feature's relevancy in the configurable system [9, 17]. However, to compute the commonality, it is necessary to be able to count the number of valid configurations.

Homogeneity. The metric describes the similarity of valid configurations induced by the configurable system. If the common base of features in different configurations is small, this might indicate that using a product line instead of single products might not pay

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
VaMoS '20, February 5–7, 2020, Magdeburg, Germany
© 2020 Copyright held by the owner/author(s). Publication rights licensed to the Association for Computing Machinery.
ACM ISBN 978-1-4503-7501-6/20/02…$15.00
https://doi.org/10.1145/3377024.3377025

off. A common way to compute homogeneity is the commonality mean over all features [14, 17].

Rating Errors. Counting the number of products can help to rate the impact of an error. After identifying an erroneous subset of the system, it is possible to compute the number of products containing this subset. An error appearing in more products might indicate a more critical error [27].

Variability Reduction. One might try to decrease the size of the configuration space by introducing further limiting constraints. However, in order to grasp the impact of such changes it is necessary to know the number of products before and after adding a new constraint [9].

Uniform Random Sampling. As it is mostly not feasible to analyze a configuration space by enumerating all configurations, it is common to create representative samples for a configurable system [33]. However, finding these samples is not easy. Randomly selecting features often results in invalid configurations. Only including the resulting valid configurations still does not guarantee randomness [35]. One way to achieve random samples is uniform random sampling [33]. The goal is to create a bijection between integers and configurations. Then, by randomly selecting an integer within the range, each configuration has the same probability to be included in the sample. The bijection can be achieved using #SAT by recursively assigning the variables [33]. For each assignment, the number of valid configurations needs to be computed [36]. This requires an efficient model counter, especially for large systems [33].

3 MOTIVATING EXAMPLE

Figure 1 shows a feature diagram representing a simplified car product line. It displays the tree structure and additional cross-tree constraints given in propositional logic. The tree structure and the cross-tree constraints limit the set of valid configurations. Each car of the product line is required to have a *Carbody*. This is indicated by the mandatory property of the feature. In contrast, a *Radio* is an optional feature. A configuration that does not contain exactly one of the *Gearbox* types, *Manual* or *Automatic*, is invalid, as they appear in an alternative-relation in the feature diagram. Furthermore, the *Ports* of a *Radio* include at least one of *USB* and *CD*. This relation is described by an or-relation. The cross-tree constraint *Navigation* $\Rightarrow$ *USB* represents that a car with *Navigation* requires an *USB*-port.

Feature diagrams are a visual representation of feature models [7]. They are commonly used to define configurable systems [7]. Each configuration corresponds to a selection of the features contained in the model. Knowing the exact number of configurations can help analyzing the feature model. For example, the feature *USB* is in 32 of the 42 (76,1%) valid configurations, while *CD* is only in 20 (47,6%). This relative share is called commonality and indicates the relevancy of a feature. During the development of the displayed car product line, it might be more effective to prioritize *Carbody* over *Radio* to build as many cars as possible with few features. Another scenario might be aiming to reduce the number of different cars induced by the product line for better maintainability. Suppose, the developers consider one of three options to reduce the number of configurations: removing *Bluetooth*, allowing only one of *USB* and *CD*, or making *Radio* a mandatory feature. Computing the number of configurations after each of the changes shows the variability

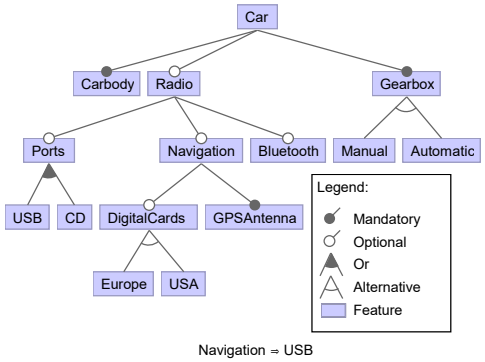


Figure 1: Example feature model adapted from Ananieva et al. [3]

reduction of each change. The resulting feature models contain 22, 26, and 40 valid configurations respectively. Therefore, removing *Bluetooth* is the most effective change regarding the reduction of variability. To enable such analyses, we are interested in computing the number of valid configurations.

Without the cross-tree constraints, computing the number of valid configurations has linear time complexity in the number of features. Only considering the tree-structure, the selections in a sub-tree are completely independent from selections in other sub-trees. Therefore, it is possible to compute the model count of each sub-tree separately. Without cross-tree constraints the model defines 66 valid configurations. This can be computed by traversing through the tree once recursively by applying rules for each relation type. For example, the number of valid configurations of an alternative sub-tree is equal to the sum of its childrens model counts. Leaf features contain exactly one configuration. Thus, the computation has linear time complexity. However, this is not possible anymore when considering cross-tree constraints.

With cross-tree constraints, the number of configurations cannot be computed in polynomial time complexity. Every feature model can be translated to a propositional formula [30]. Furthermore, a feature model that contains cross-tree constraints can represent every propositional formula [25]. Thus, computing the satisfiability of a model with those constraints is at least as hard as SAT. Determining whether a formula is satisfiable after computing the number of valid assignments is trivial. Therefore, #SAT is at least as hard as SAT [23]. It follows that the complexity of counting the number of valid configurations with cross-tree constraints is not polynomial.

4 MODEL COUNTING

SAT solvers are used to determine whether a propositional formula is satisfiable (i.e., there is an assignment with which the formula evaluates to true) [20]. Let F be a propositional formula and $vars(F)$ with $|vars(F)| = n$ the corresponding set of variables. An assignment is a function $\alpha : vars(F) \rightarrow \{0, 1\}$ that maps variables contained in F to the truth values 0 or 1 [27]. Assignments can be partial, meaning that not every variable $v \in vars(F)$ is mapped to 0 or 1. Otherwise, the assignment is called full [27]. The cardinality $|\alpha| \leq n$ corresponds to the number of variables mapped to 0 or 1 in

α. The function $F(\alpha) \rightarrow \{0,1\}$ evaluates whether a full assignment α satisfies the formula F. We call an assignment α with $F(\alpha) = 1$ a valid assignment or a model.

Propositional model counting (for short #SAT) is defined as computing the number of valid full assignments of a propositional formula [18, 27]. $\#F = |\{\alpha | F(\alpha) = 1\}|$ corresponds to the number of models (i.e., valid full assignments of formula F).

#SAT is widely believed to be harder to solve than SAT [12]. #SAT is obviously at least as hard as SAT, because it is trivial to determine whether there is at least one solution after computing the number of solutions [23]. Even though the problem is difficult with regard to the theoretical complexity, the #SAT community made significant advances in the last decade [6, 12, 15, 28, 42]. We focus on the major model counting methods, namely DPLL-based [6, 10, 12, 39, 42], d-DNNF-based [15, 28, 32], and BDD-based [43] counting. We give a short introduction to those three methods in the following.

Davis-Putnam-Logemann-Loveland (DPLL)-based counting uses an explorative search through the tree representing the formula with n variables. Each depth in the tree corresponds to the assignment of one variable of the formula. The goal is to find an assignment α with $|\alpha| \leq n$ that either satisfies or does not satisfy the formula for each possible assignment of the remaining $n - |\alpha|$ variables. If the formula evaluates to false under the assignment α, the number of models for the corresponding branch is 0. If it evaluates to true, the number of models for the corresponding branch is $2^{n-|\alpha|}$, which is the number of possible assignments of the remaining variables. A valid full assignment has exactly $2^{n-n} = 1$ model. After computing a result for a branch, DPLL uses backtracking to find remaining assignments. This is done until the number of models for every branch is known. The sum of computed results is the exact number of models [11].

Another possible way to compute number of models are d-DNNF compilers. The term d-DNNF stands for deterministic, decomposable negation normal form. d-DNNF is a subset of NNF that satisfies determinism and decomposability [16]. A formula L is called *deterministic* if each child $D_1, ..., D_n$ of a disjunction $D \in L$ is logically disjunct (i.e., $\forall i, j : i \neq j : D_i \wedge D_j \models \bot$) [16]. A formula is called *decomposable* if the children $C_1, ..., C_n$ of a conjunction C share no variables (i.e., $\forall i, j : i \neq j : vars(C_i) \cap vars(C_j) = \emptyset$) [16]. Determinism implies that the children $D_1, ..., D_n$ of a disjunction D share no common solutions. Therefore, the model count of the sub-tree corresponding to the disjunction is equal to the sum of the children's results (i.e., $\#D = \sum_{i=1}^{n} \#D_i$) [11]. Decomposability implies that each conjunction C can be decomposed into disjoint sub-trees representing the children $C_1, ..., C_n$. It follows that the model count of the conjunction is equal to the product of the results for each subtree (i.e., $\#C = \prod_{i=1}^{n} \#C_i$) [11]. Using both properties, it is possible to compute the overall number of solutions by traversing through the formula once [11]. A d-DNNF-based model counting corresponds to compiling a propositional formula to d-DNNF. A common way to compile a CNF to d-DNNF is an exhaustive DPLL algorithm [15, 32]. After the compilation, computing the model count takes polynomial time [15]. The d-DNNF is strictly more succinct than binary decision diagramms [16]. Therefore, d-DNNFs should take less memory to express a propositional formula.

A binary decision diagram (BDD) is a directed acyclic graph that represents a propositional formula F [19]. Each BDD contains two terminal nodes, with values 0 and 1, and at least one variable node. Every variable node corresponds to one variable $v \in vars(F)$ and has exactly two child nodes, one high and one low child. Each child can either be another variable node or a terminal node. Suppose an assignment α, it can be evaluated by recursively traversing through the BDD. At each variable node corresponding to variable v, the next edge taken by the traversal depends on $\alpha(v)$. Depending on whether $\alpha(v) = 0$ or $\alpha(v) = 1$, the edge to the low or high child is taken respectively. The assignment α satisfies the formula if the resulting path ends in the 1-terminal node [19]. The required space for a BDD is highly dependent on the variable ordering [31, 34]. Therefore, the literature often considers ordered binary decision diagrams (OBDD) and reduced OBDDs [19, 22, 31]. A BDD is called *ordered* if the variable index of a parent is always higher than the indices of its children [17]. Additionally, a BDD is called *reduced* if the two following properties hold. First, there is no variable node whose low and high child are the same node. Second, there are no two distinct nodes for which the ancestors are equivalent [43]. BDDs can be compiled from a propositional formula using an exhaustive DPLL search as d-DNNF [21, 22]. The model count can be determined by traversing through the diagram starting from the 1-terminal node. This improves the runtime as unsatisfied paths are not considered. Furthermore, the efficiency of computing the number of valid assignments can be improved by using dynamic programming [17].

5 EVALUATION

In this section, we report on our evaluation of #SAT solvers on industrial feature models. We examine the scalability of #SAT solvers, the correlation of size of a system with the scalability, and the evolution of configurable systems.

5.1 Evaluated #SAT Solvers

Our benchmark considers three types of model counters. Mainly, we evaluated DPLL-based solvers and CNF to d-DNNF compilers. Additionally, we added one CNF to BDD compiler for comparison, as previous works also used BDDs for model counting [2, 19, 29]. Every chosen solver uses CNFs in DIMACS-format as input and is publicly available. Furthermore, we only evaluated exact #SAT solvers, contrary to approximate ones.

- DPLL-based solvers: Relsat [6], Cachet [39], sharpSAT [42], countAntom [12], picoSAT [10]
- CNF to d-DNNF compilers: c2d [15], d4 [28], dSharp [32]
- CNF to BDD compiler: CNF2OBDD [43]

5.2 Subject Systems

We evaluate the performance of the listed #SAT solvers on industrial configurable systems from the automotive and operating system domain. First, we analyze configurable systems provided by Knüppel et al. [25]. These systems were extracted by snapshots of an automotive product line and translating KConfig and CDL models. KConfig was designed for managing Linux configurations and CDL for managing eCos [25]. The systems are available as FeatureIDE

Subject Systems	#Models	#Features	#Constraints
KConfig	7	96-6467	14-3545
CDL	116	1178-1408	816-956
Automotive02	4	14010-18616	666-1369
Automotive03	5	149-588	0-1184
Automotive04	50	127-531	0-623
Automotive05	136	246-1674	0-11632

Table 1: Number of models, range of features, and range constraints of the evaluated subject systems

feature models.[1] We converted the models to DIMACS-format using FeatureIDE 3.5.5.[2]

Second, we were given access to industrial models for three different systems from the automotive domain. These models were provided in a proprietary format. With the help of company interns, we translated their data structure into feature models. Afterwards, we transformed the resulting models to the DIMACS-format using FeatureIDE 3.5.5. Each feature and constraint in the system included a date which indicated the introduction into the system. Furthermore, some included a date that indicated exclusion. Using these properties, we created snapshots that represented the system only with features and constraints that are included at the given timestamp. In the remainder, we distinguish between the entire system and snapshots of that system. We refer to these snapshots as models. Furthermore, if we use the term system, we always refer to the latest model of that system. Table 1 presents properties of all systems considered in the experiment. All models and systems only contain boolean variables and boolean constraints.

5.3 Experiment Design

In the first stage, we evaluated all solvers specified in Section 5.1 on every subject system described in Section 5.2. If a solver was not able to compute a result within 10 minutes, the process was terminated. In the second stage, we repeated the experiment with all the systems and models that could not be evaluated within 10 minutes by any of the solvers. Here, the timeout was set to 24 hours. For 127 systems, a single solver that does not scale might need more than four months for the computation. Therefore, we excluded all solvers in the second stage that could not evaluate at least half of the models within 10 minutes in the first stage.

All solvers except CNF2OBDD provided parameters to internally limit the maximum memory used. During the entire benchmark, we set this memory limit for each solver to 8 GB. However, sometimes solvers exceeded the given limit. If a solver reached 10 GB of memory, the process was terminated to prevent an unintended use of memory. The solver countAntom allows multi-threading and the maximum number of used threads can be defined. During the first stage, we evaluated countAntom with one and four available threads separately for better comparison. For the second stage, we only considered countAntom with four available threads. During the remainder of the evaluation, we consider the experiment with four threads if not stated otherwise.

[1] https://github.com/AlexanderKnueppel/is-there-a-mismatch
[2] https://github.com/FeatureIDE/FeatureIDE

5.4 Results

Figure 2 shows the runtime of each solver on all configurable systems listed in Section 5.2. The runtime is logarithmic. The red line indicates the timeout and the blue line indicates that a solver reached the memory limit. In the first stage, Picosat reached the timeout on every system and Relsat for 119 of the 127 (93.7%) systems. CNF2OBDD scaled to only 3 of 127 systems (2.36%), namely axTLS (96 features), uClinux-base (380), and Automotive04 (531). It reached the memory limit of 8 GB for 122 and the timeout for 2 systems. Most other solvers only reached timeout for Linux and Automotive05. The only other exceptions were Cachet and c2d which both had an additional timeout on one other model. countAntom was the fastest solver for 113 systems. countAntom with only one thread was 1.9 times slower on average than with four threads. However, it was still the fastest solver on all these 113 systems. dSharp was terminated for one system for reaching the hard memory limit of 10 GB.

In the second stage, the experiment was repeated without Picosat, Relsat, and CNF2OBDD with a timeout of 24 hours to examine whether a solver is able to compute the remaining systems. However, even within 24 hours not a single solver was able to compute a result for Linux or the latest model of Automotive05.

Figure 3 shows the runtimes in seconds of the six remaining solvers on the 5 Automotive03, 50 Automotive04, and 136 Automotive05 models. In each diagram, the runtime has a logarithmic scale. For Automotive03 and Automotive04, c2d required up to 5.7 seconds to evaluate the models. None of the other solvers remaining in the second stage needed more than 250 milliseconds for any of these models. Cachet and sharpSAT even computed the number of products within 50 milliseconds for the models of Automotive03 and Automotive04. The following solvers were the fastest ones on at least one of these models: sharpSAT (fastest for 29 models), Cachet (24), and dSharp (1).

While the remaining solvers scaled for every model of Automotive03 and Automotive04, this is not the case for Automotive05. The model count of 33 of the 136 models could be computed by at least one solver within 10 minutes. Due a lack of time, the experiment was repeated with a 24 hours timeout with only the three solvers that solved the highest number of models: c2d (33 solved models), countAntom (31/ 23 with one thread), and d4 (29). Within 24 hours, the number of configurations could be computed for 62 models by c2d (51 solved models), countAntom (62), and d4 (49). All of those 62 models were within the 65 earliest ones. For the 65th model 21.6 hours were required to compute the number of configurations.

Figure 4 shows the correlation of the number of features, constraints, and clauses with the runtime of the fastest solver. In each diagram, both scales are logarithmic. Every system with fewer than 1,000 features, 1,000 constraints, or 10,000 clauses was evaluated within 100 milliseconds. The two systems that reached the timeout, the Linux product line and Automotive05, contain 6,467 and 1,663 features. The number of configurations for Automotive02 which contains 18,616 features, 1,369 constraints, and 350,221 clauses was computed within 500 milliseconds.

Figure 5 shows the runtime of the fastest solver for Automotive03, Automotive04, and Automotive05 in correlation to number

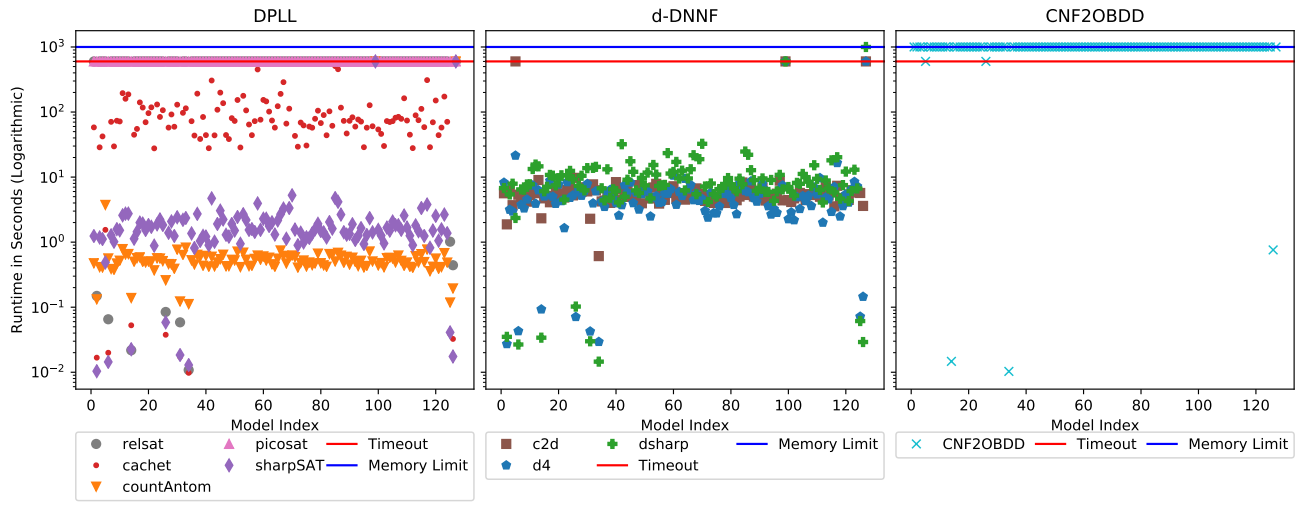


Figure 2: Runtime in seconds (logarithmic) of all solvers on every system with a timeout of 10 minutes

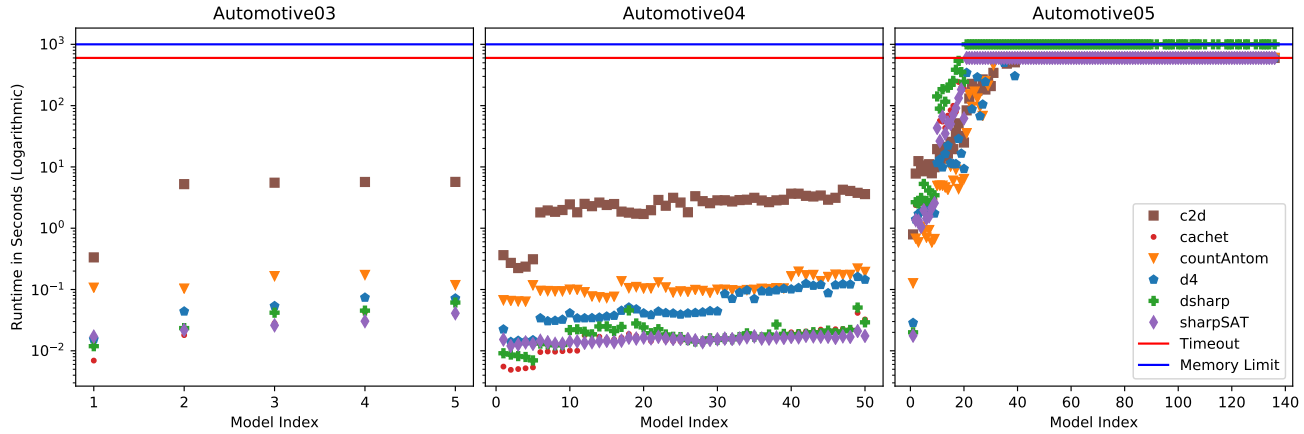


Figure 3: Runtime in seconds of remaining solvers on every model of Automotive03, Automotive04, and Automotive05 with a timeout of 10 minutes

of features, constraints, and clauses. The three systems are distinguished by means of different colors. In each diagram, the runtime has a logarithmic scale. Every model with less than 1,000 features, 3,000 constraints, or 10,000 clauses was evaluated within 100 milliseconds.

The computed number of products in correlation to the number of features systems are displayed in Figure 6 for every evaluated model. Both scales are logarithmic. The different systems are distinguished by means of different colors. The results of Automotive02 are omitted for better readability. Table 2 shows the ranges of number of configurations of each system, including Automotive02. For every evaluated system that contains multiple models, the number of features is weakly monotonically increasing. This is not the case for the number of configurations as sometimes a later model has fewer configurations. However, overall the configuration space of every automotive system is also growing over time. Table 2 shows the ranges of number of configurations for every evaluated system.

Subject System	#Models(Solved)	#Configurations
KConfig	7(6)	$8.3 * 10^{11} - 2.1 * 10^{201}$
CDL	116(116)	$2.6 * 10^{118} - 3 * 10^{136}$
Automotive02	4(4)	$4.7 * 10^{1260} - 1.7 * 10^{1534}$
Automotive03	5(5)	$5.8 * 10^{28} - 2.5 * 10^{31}$
Automotive04	50(50)	$1.6 * 10^{13} - 4.8 * 10^{23}$
Automotive05	136(62)	$2.2 * 10^{48} - 1.1 * 10^{66}$

Table 2: Number of models and range of number of valid configurations

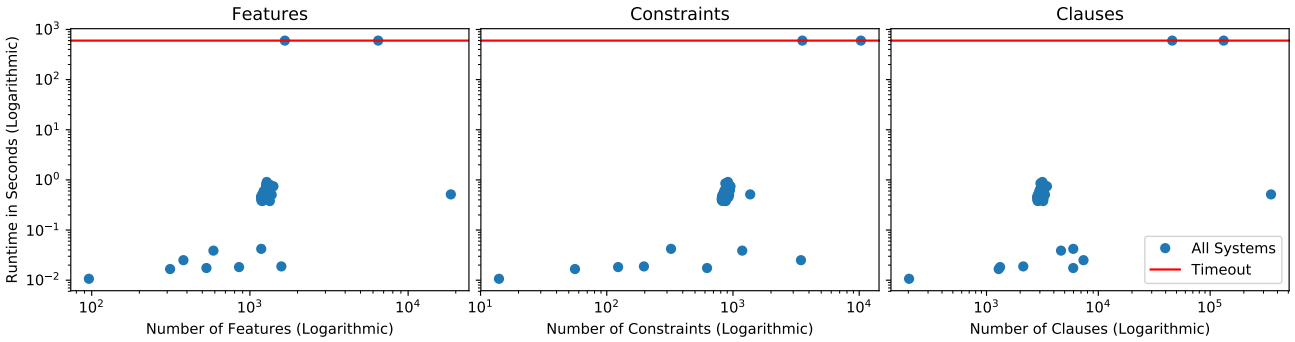


Figure 4: Correlation of number of features, constraints, and clauses with the runtime of the fastest solver on every system

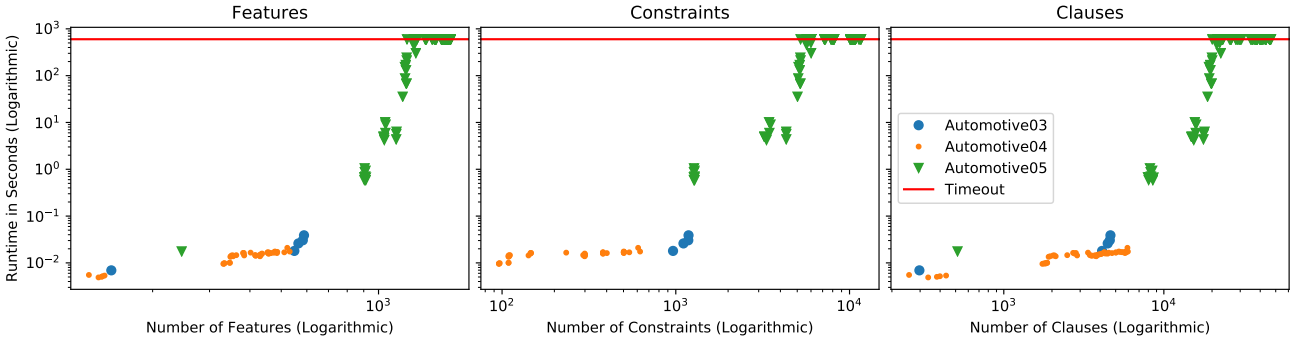


Figure 5: Correlation of number of features, constraints, and clauses with the runtime of the fastest solver for every model of Automotive03, Automotive04, and Automotive05

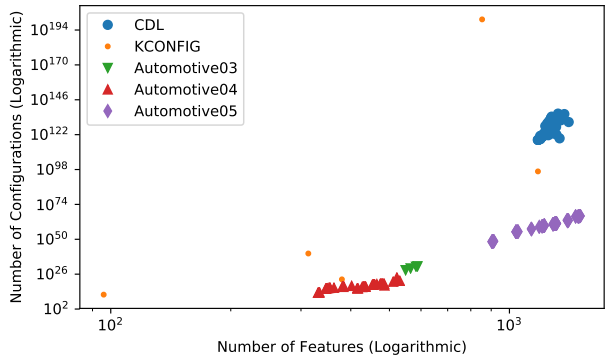


Figure 6: Correlation between number of features and number of configurations for all systems

5.5 Discussion

In this section, we discuss the results regarding our five research questions **RQ1-RQ5**.

RQ1: Do #SAT solvers scale to industrial configuration spaces? Our results indicate that it is possible to apply #SAT solvers to analyze various configurable systems. Overall, the model count was computed for 125 of the 127 analyzed systems and all models of Automotive02-Automotive04 within 10 minutes. However, the current solvers do not scale for all industrial systems that were available to us. Two systems could not be evaluated: Automotive05 and Linux. Out of the 136 Automotive05 models only 35 were evaluated within 10 minutes. With a 24 hours timeout, 62 models were evaluated by at least one solver.

RQ2: Is one #SAT solver superior to the other solvers? None of the solvers was superior to the other solvers in every instance. Cachet is the fastest solver for 24 out of the 55 models from Automotive03 and Automotive04, but only for 7 other systems. c2d solved the highest number of models from Automotive05 of all solvers within 10 minutes, but required by far the longest runtime for the Automotive03 and Automotive04 models of the solvers considered in the second stage. However, countAntom was the fastest one in 113 of the 127 systems that were solved by at least one solver within 10 minutes. Furthermore, the solver computed the highest number of models from Automotive05 within 24 hours.

Even though the best performing solver countAntom is DPLL-based, the results do not indicate a superiority of DPLL-based over d-DNNF-based solvers regarding the scalability of computing the number of configurations. However, for larger systems the BDD-based solver has scalability issues, as the memory limit was reached for 122 systems (i.e., 96.1% of all systems). picoSAT is the only solver

that computes the number of valid solutions by enumerating them and scaled to none of the evaluated systems.

RQ3: Does the runtime of the solvers correlate to the size or complexity of the configuration space? The results indicate that it is always easy to compute the number of configurations for small and incomplex systems. For example, every system with less than 1,000 features or constraints was evaluated within 100 milliseconds. However, a large number of features, constraints, or clauses does not necessarily cause a time-consuming computation. The Automotive02 system contains by far the most features, but sharpSAT still evaluated it in less than 500 milliseconds. Outside of the given experiment design, we created a model of Automotive05 that included every feature and constraints that appears in the product line at any point in time. This model contains 27,532 constraints. Still, the fastest solver did not even need 2 seconds for the evaluation, while reaching timeout for models with less than 10,000 constraints.

Nevertheless, the scalability of different versions of a single system seems to positively correlate with the size and complexity. This is indicated by the benchmarks for Automotive03, Automotive04, and Automotive05 in Figure 5.

RQ4: How does the number of valid configurations relate to the number of all configurations? A model with n features contains up-to 2^n configurations [9]. However, a larger n does not guarantee a larger number of valid configurations. Busybox contains 854 features and induces $2.1 * 10^{201}$ valid configurations while the largest model from Automotive05 that could be evaluated contains 1506 features and induces $1.1 * 10^{66}$ valid configurations. Nevertheless, the results for Automotive03, Automotive04, and Automotive05 indicate that within similiar systems there is a positive correlation between all and valid configurations.

RQ5: How does the number of valid configurations change during the evolution of a configurable system? During our experiments only four configurable systems contained multiple models: Automotive02-Automotive05. For each of those systems the number of features monotonically increased at every new model. The results also indicate that configuration spaces of configurable systems continually grow. However, there are cases where a newer model induced fewer configurations. In these cases, the corresponding update introduced new constraints and only few new features.

In order to improve the scalability of #SAT solvers on configurable systems, we consider two possibilities. On the one hand, the #SAT community pushes the limits by further improving solvers. On the other hand, even continually improved solvers might not scale for large configurable systems. Another way to improve scalability, might be to limit the configuration space by reducing the variability (e.g., with feature-model slicing [1, 4, 26] or by modularity principles [40]).

5.6 Threats to Validity

During the evaluation of #SAT solvers on industrial feature models, we identified a few potential threats to validity.

Translating the subject systems to feature models. Our translation of the automotive product lines might be incorrect. However, we created the parser in direct cooperation with company interns. Furthermore, the resulting model was reviewed by them. Another threat is that we could not parse a few constraints as they contained features that did not appear in the feature model. Therefore, we possibly reduced the complexity of the resulting models.

Knüppel et al. also remarked some threats to internal validity regarding their translation of configurable systems [25]. First, there are differences between CDL and KConfig and the target format. Second, the translation may have removed a few cross-tree constraints. Furthermore, a few cases lead to features that did not appear in the input format [25].

Translating feature models to the DIMACS-format. For the translation to DIMACS-format, we used the FeatureIDE-library [24]. In order to ensure a correct translation, we performed the following sanity checks. First, we manually computed the model count of small feature models and compared these results with the ones computed by the solvers. Second, we made changes to the feature model that should change the model count in a certain way. For example, we added an optional feature to the root which should always double the number of products. Another important aspect of the translation to CNF is that the model count of the new formula has to stay the same. This is not given for every conversion method [27]. However, FeatureIDE uses a transformation that does not introduce new variables, nor changes the number of solutions. The results of the sanity checks support this claim.

Computation of the solvers. We only used external solvers without a possibility of directly verifying the results. However, for every computed instance the model count returned by each solver was equal. This is a strong indicator for the correctness of the solvers.

External Validity. We can not claim that our results can be transferred to all other industrial configurable systems. The product lines provided by our industry partner are limited to one domain and company. Furthermore, Knüppel et al. also argued that their results can not directly be transferred to other real-world systems [25]. However, overall we considered multiple domains and various different systems. Therefore, our results possibly indicate the scalability of #SAT solvers for other configurable systems.

6 RELATED WORK

Model Counting of Configurable Systems. Kübler et al. [27] also evaluated the use of two older #SAT solvers, Cachet [38] and c2d [15], on three different body styles of a automotive product line. We evaluated both solvers and they were outperformed by newer solvers on most instances. However, the authors also proposed their own model counter that was not based on conjunctive normal form and performed well even on larger systems. We were not able to acquire their solver or their evaluated product lines. Therefore, we could not directly compare the results.

Pohl et al. [37] evaluated different feature model analyses including model counting using binary decision diagrams, constraint satisfaction problem solvers, and SAT solvers. However, the authors used models with much smaller configuration spaces and fewer features for their evaluation. Their analyzed configuration spaces only reached up to 10^8 valid configurations.

Current Tool Support. BDDs are a popular choice for counting the number of products of configurable systems [2, 19, 29]. The benchmark of Pohl et al. indicated that BDDs might be faster than

#SAT solvers [37]. Furthermore, it is possible to compute the BDD offline and answer queries on the live system in polynomial time [19]. However, our results indicate that BDDs need too much memory for larger feature models. Additionally, d-DNNFs can be computed offline as well and are stricly more succinct than BDDs [16].

FeatureIDE uses a regular SAT solver to compute the number of valid configurations [41]. The tool realizes counting with a regular SAT solver using blocking clauses; after finding a valid assignment α, it is negated and added as a clause to the formula. Thus, α is not a valid assignment for the resulting formula and the next run of the solver returns another assignment. For each found assignment (i.e., valid configuration), an execution from scratch is required. Therefore, the algorithm does not scale for larger systems [43].

Non-propositional Model Counting. Constraint satisfaction problems (CSP) are an alternative to propositional logic for the representation of feature models [7–9, 37]. CSPs are defined by a set of variables, domains for each variable, and constraints over these variables. For CSPs, the variables may also be integers or intervals contrary to propositional boolean variables which are strictly binary [7]. Benavides et al. used constraint programming (CP) to compute the number of valid configurations of feature models. However, the models considered in their experiment only included up to 23 features [9]. The results of the experiment of Pohl et al. which compared SAT solvers, BDDs, and CSP solvers also indicated that the analyzed CSP solvers scale far worse than the #SAT solvers evaluated in our experiment [37]. Munoz examined counting the number of valid configurations of feature models with numerical features for uniform random sampling. The authors evaluated an SMT solver, a CP solver, and the #SAT solver sharpSAT. The numerical values were translated to propositional logic using bit-blasting [33]. In their experiment, sharpSAT vastly outperformed the CP and SMT solver.

7 CONCLUSION

Managing the variability of configurable systems is hard without computing the number of valid configurations. In this paper, we analyzed three d-DNNF-based [15, 28, 32], five DPLL-based [6, 10, 12, 39, 42], and one BDD-based [43] #SAT solver on industrial configurable systems. Those systems consisted of models provided by Knüppel et al. [25] and three automotive product lines provided by our industry partner. Most of the features and constraints of the automotive product lines were only valid for a specific period of time. Therefore, we were able to create snapshots, each representing one timestamp for each product line.

Out of the 127 evaluated systems, 125 could be computed within 10 minutes by at least one solver. However, the remaining two systems, Linux and a majority of the snapshots of the largest automotive product line, could not even be computed within 24 hours. This indicates that current #SAT solvers scale to many, but not to all systems. Furthermore, the results for different snapshots of the same system indicate that the configuration space of these systems is continually growing during the evolution. While the #SAT community keeps pushing the limits for their solvers, even new solvers do not necessarily scale for the growing systems. Deliberately reducing the variability of a configurable system might be necessary to enable the computation of the number of valid configurations.

In our experiments, some solvers were able to evaluate only a minority, if any, of the given systems. While, none of the solvers was strictly superior to all other ones, countAntom [12] was the fastest solver for 89.0% of the evaluated systems. Nevertheless, it might be beneficial to use different solvers in parallel.

To measure the complexity of a system, we considered the number of clauses and constraints. For the size, we considered the number of features. The results indicated that the number of valid configurations of a more complex or larger system does not necessarily take more time to be computed. However, the runtime has a positive correlation with the size and complexity within similar systems. Furthermore, a majority of the solvers scales for all smaller or less complex models.

8 FUTURE WORK

In this section, we describe further tasks in counting the configuration space of configurable systems.

Approximate #SAT Solvers. In this paper, we focused on exact model counting. However, the results indicated that exact model counters may not be scalable for large industrial systems. One solution for this problem might be approximate #SAT solvers [13, 18]. These aim to count the number of satisfying assignments within a given confidence level [13]. Approximate solvers often allow various options to parameterize the process of model counting [13, 18]. For approximated analysis of configurable systems it is necessary to find suitable parameters that result in efficient and effective computations.

Further Metrics for a Meta-Solver. Our results show that the solvers perform differently depending on the system. None of the solvers was strictly superior to the others. Analyzing the meta-data of configurable systems might enable an efficient meta-solver that selects the most promising solver depending on a given instance. For regular SAT, it is already known that selecting a solver based on single instances vastly improves the performance [44].

Directly Translate Feature Models to d-DNNF. Every experiment used propositional formulas in conjunctive normal form. The translation to CNF was not considered in the runtime. However, for the larger systems, the translation requires a considerable amount of time. Directly translating the feature model to d-DNNF might result in two benefits. First, the time overhead of computing the model to CNF would be eliminated. Second, it is possible that the direct translation is even less expensive itself.

Analyze Memory Usage. We mainly focused on the runtime of the solvers. However, memory usage plays an important role as well. Especially, the d-DNNF and BDD compilers need a high amount of memory to translate the formula.

Growth of Configuration Spaces. Our results indicated that the size of configurable systems we analyzed grow over time. However, we do not have enough data to translate this observation on other systems. Analyzing how fast configuration spaces are growing might enable the following estimations. First, the growth might indicate whether #SAT solvers scale for a future model. Second, the number of valid configurations of a future model can possibly be estimated without using a #SAT solver.

REFERENCES

[1] Mathieu Acher, Philippe Collet, Philippe Lahire, and Robert B France. 2011. Slicing feature models. In *Proceedings of the 2011 26th IEEE/ACM International Conference on Automated Software Engineering*. IEEE Computer Society, 424–427.

[2] Mathieu Acher, Philippe Collet, Philippe Lahire, and Robert B France. 2013. Familiar: A domain-specific language for large scale management of feature models. *Science of Computer Programming* 78, 6 (2013), 657–681.

[3] Sofia Ananieva. 2016. *Explaining Defects and Identifying Dependencies in Interrelated Feature Models.* Ph.D. Dissertation. Institute of Software.

[4] Sofia Ananieva, Matthias Kowal, Thomas Thüm, and Ina Schaefer. 2016. Implicit constraints in partial feature models. In *Proceedings of the 7th International Workshop on Feature-Oriented Software Development*. ACM, 18–27.

[5] Ebrahim Bagheri, Tommaso Di Noia, Dragan Gasevic, and Azzurra Ragone. 2012. Formalizing interactive staged feature model configuration. *Journal of Software: Evolution and Process* 24, 4 (2012), 375–400.

[6] Roberto J Bayardo Jr and Joseph Daniel Pehoushek. 2000. Counting models using connected components. In *AAAI/IAAI*. 157–162.

[7] David Benavides, Sergio Segura, and Antonio Ruiz-Cortés. 2010. Automated analysis of feature models 20 years later: A literature review. *Information Systems* 35, 6 (2010), 615–636.

[8] David Benavides, Sergio Segura, Pablo Trinidad, and Antonio Ruiz-Cortés. 2005. Using Java CSP solvers in the automated analyses of feature models. In *International Summer School on Generative and Transformational Techniques in Software Engineering*. Springer, 399–408.

[9] David Benavides, Pablo Trinidad, and Antonio Ruiz-Cortés. 2005. Automated reasoning on feature models. In *International Conference on Advanced Information Systems Engineering*. Springer, 491–503.

[10] Armin Biere. 2008. PicoSAT essentials. *Journal on Satisfiability, Boolean Modeling and Computation* 4 (2008), 75–97.

[11] Armin Biere, Marijn Heule, and Hans van Maaren. 2009. *Handbook of satisfiability*. Vol. 185. IOS press.

[12] Jan Burchard, Tobias Schubert, and Bernd Becker. 2015. Laissez-faire caching for parallel# SAT solving. In *International Conference on Theory and Applications of Satisfiability Testing*. Springer, 46–61.

[13] Supratik Chakraborty, Kuldeep S Meel, and Moshe Y Vardi. 2013. A scalable approximate model counter. In *International Conference on Principles and Practice of Constraint Programming*. Springer, 200–216.

[14] Paul C Clements, John D McGregor, and Sholom G Cohen. 2005. *The structured intuitive model for product line economics (SIMPLE).* Technical Report. CARNEGIE-MELLON UNIV PITTSBURGH PA SOFTWARE ENGINEERING INST.

[15] Adnan Darwiche. 2004. New advances in compiling CNF to decomposable negation normal form. In *Proceedings of the 16th European Conference on Artificial Intelligence*. Citeseer, 318–322.

[16] Adnan Darwiche and Pierre Marquis. 2002. A knowledge compilation map. *Journal of Artificial Intelligence Research* 17 (2002), 229–264.

[17] David Fernandez-Amoros, Ruben Heradio, Jose A Cerrada, and Carlos Cerrada. 2014. A scalable approach to exact model and commonality counting for extended feature models. *IEEE Transactions on Software Engineering* 40, 9 (2014), 895–910.

[18] Carla P Gomes, Ashish Sabharwal, and Bart Selman. 2006. Model counting: A new strategy for obtaining good bounds. In *AAAI*. 54–61.

[19] Tarik Hadzic, Sathiamoorthy Subbarayan, Rune M Jensen, Henrik R Andersen, Jesper Møller, and Henrik Hulgaard. 2004. Fast backtrack-free product configuration using a precompiled solution space representation. *small* 10, 1 (2004), 3.

[20] Ruben Heradio, David Fernandez-Amoros, Jose A Cerrada, and Ismael Abad. 2013. A literature review on feature diagram product counting and its usage in software product line economic models. *International Journal of Software Engineering and Knowledge Engineering* 23, 08 (2013), 1177–1204.

[21] Jinbo Huang and Adnan Darwiche. 2004. Using DPLL for efficient OBDD construction. In *International Conference on Theory and Applications of Satisfiability Testing*. Springer, 157–172.

[22] Jinbo Huang and Adnan Darwiche. 2005. DPLL with a trace: From SAT to knowledge compilation. In *IJCAI*, Vol. 5. 156–162.

[23] David S Johnson. 1992. The NP-completeness column: an ongoing guide. *Journal of algorithms* 13, 3 (1992), 502–524.

[24] Christian Kastner, Thomas Thum, Gunter Saake, Janet Feigenspan, Thomas Leich, Fabian Wielgorz, and Sven Apel. 2009. FeatureIDE: A tool framework for feature-oriented software development. In *Proceedings of the 31st International Conference on Software Engineering*. IEEE Computer Society, 611–614.

[25] Alexander Knüppel, Thomas Thüm, Stephan Mennicke, Jens Meinicke, and Ina Schaefer. 2017. Is there a mismatch between real-world feature models and product-line research?. In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*. ACM, 291–302.

[26] Sebastian Krieter, Reimar Schröter, Thomas Thüm, Wolfram Fenske, and Gunter Saake. 2016. Comparing algorithms for efficient feature-model slicing. In *Proceedings of the 20th International Systems and Software Product Line Conference*. ACM, 60–64.

[27] Andreas Kübler, Christoph Zengler, and Wolfgang Küchlin. 2010. Model counting in product configuration. *arXiv preprint arXiv:1007.1024* (2010).

[28] Jean-Marie Lagniez and Pierre Marquis. 2017. An Improved Decision-DNNF Compiler.. In *IJCAI*. 667–673.

[29] Marcilio Mendonca, Moises Branco, and Donald Cowan. 2009. SPLOT: software product lines online tools. In *Proceedings of the 24th ACM SIGPLAN conference companion on Object oriented programming systems languages and applications*. ACM, 761–762.

[30] Marcilio Mendonca, Andrzej Wąsowski, and Krzysztof Czarnecki. 2009. SAT-based analysis of feature models is easy. In *Proceedings of the 13th International Software Product Line Conference*. Carnegie Mellon University, 231–240.

[31] Marcilio Mendonca, Andrzej Wasowski, Krzysztof Czarnecki, and Donald Cowan. 2008. Efficient compilation techniques for large scale feature models. In *Proceedings of the 7th international conference on Generative programming and component engineering*. ACM, 13–22.

[32] Christian Muise, Sheila McIlraith, J Christopher Beck, and Eric Hsu. 2010. Fast d-DNNF Compilation with sharpSAT. In *Workshops at the twenty-fourth AAAI conference on artificial intelligence*.

[33] Daniel-Jesus Munoz, Jeho Oh, Mónica Pinto, Lidia Fuentes, and Don Batory. 2019. Uniform random sampling product configurations of feature models that have numerical features. In *Proceedings of the 23rd International Systems and Software Product Line Conference-Volume A*. ACM, 39.

[34] Macha Nikolskaia and Antoine Rauzy. 1998. Heuristics for BDD handling of sum-of-products formulae. In *Proceedings of the European Safety and Reliability Association Conference, ESREL'98*. Citeseer.

[35] Jeho Oh, Don Batory, Margaret Myers, and Norbert Siegmund. 2017. Finding near-optimal configurations in product lines by random sampling. In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*. ACM, 61–71.

[36] Jeho Oh, Paul Gazzillo, Don Batory, Marijn Heule, and Maggie Myers. 2019. Uniform sampling from kconfig feature models. *The University of Texas at Austin, Department of Computer Science, Tech. Rep. TR-19-02* (2019).

[37] Richard Pohl, Kim Lauenroth, and Klaus Pohl. 2011. A performance comparison of contemporary algorithmic approaches for automated analysis operations on feature models. In *Proceedings of the 2011 26th IEEE/ACM International Conference on Automated Software Engineering*. IEEE Computer Society, 313–322.

[38] Tian Sang, Fahiem Bacchus, Paul Beame, Henry A Kautz, and Toniann Pitassi. 2004. Combining Component Caching and Clause Learning for Effective Model Counting. *SAT* 4 (2004), 7th.

[39] Tian Sang, Paul Beame, and Henry Kautz. 2005. Heuristics for fast exact model counting. In *International Conference on Theory and Applications of Satisfiability Testing*. Springer, 226–240.

[40] Reimar Schröter, Sebastian Krieter, Thomas Thüm, Fabian Benduhn, and Gunter Saake. 2016. Feature-model interfaces: the highway to compositional analyses of highly-configurable systems. In *Proceedings of the 38th International Conference on Software Engineering*. ACM, 667–678.

[41] Thomas Thum, Christian Kastner, Sebastian Erdweg, and Norbert Siegmund. 2011. Abstract features in feature modeling. In *2011 15th International Software Product Line Conference*. IEEE, 191–200.

[42] Marc Thurley. 2006. sharpSAT–counting models with advanced component caching and implicit BCP. In *International Conference on Theory and Applications of Satisfiability Testing*. Springer, 424–429.

[43] Takahisa Toda and Takehide Soh. 2016. Implementing efficient all solutions SAT solvers. *Journal of Experimental Algorithmics (JEA)* 21 (2016), 1–12.

[44] Lin Xu, Frank Hutter, Holger H Hoos, and Kevin Leyton-Brown. 2008. SATzilla: portfolio-based algorithm selection for SAT. *Journal of artificial intelligence research* 32 (2008), 565–606.

C.5 SPLC'24: Collecting Feature Models from the Literature: A Comprehensive Dataset for Benchmarking

Published under Creative Commons Attribution 4.0 International (CC-BY), see https://creativecommons.org/licenses/by/4.0/.

[B3] Chico Sundermann, Vincenzo Francesco Brancaccio, Elias Kuiter, Sebastian Krieter, Tobias Heß, and Thomas Thüm. "Collecting Feature Models from the Literature: A Comprehensive Dataset for Benchmarking". In: *Proc. Int'l Systems and Software Product Line Conf. (SPLC)*. ACM, 2024, pp. 54–65. DOI: 10.1145/3646548.3672590





Collecting Feature Models from the Literature: A Comprehensive Dataset for Benchmarking

Chico Sundermann
University of Ulm
Germany

Vincenzo Francesco Brancaccio
University of Ulm
Germany

Elias Kuiter
University of Magdeburg
Germany

Sebastian Krieter
Paderborn University
Germany

Tobias Heß
University of Ulm
Germany

Thomas Thüm
Paderborn University
Germany

ABSTRACT

Feature models are widely used for specifying the valid configurations of product lines. Many automated analyses on feature models have been considered, but they often depend on computationally complex algorithms (e.g., solving satisfiability problems). To identify and develop efficient reasoning engines, it is necessary to compare their performance on practically relevant feature models. However, empirical evaluations on feature-model analysis often suffer from the limitations of available feature-model datasets in terms of transferability. A major problem is the accessibility of relevant feature models as they are scattered over numerous publications. In this work, we perform a literature survey on empirical evaluations that target the performance of feature-model analyses to examine common evaluation practices and collect feature models for future evaluations. Furthermore, we examine the suitability of the derived collection for benchmarking performance. To improve accessibility, we provide a repository including all 2,518 identified feature models from 13 application domains, such as system software.

CCS CONCEPTS

• **Software and its engineering → Software product lines**.

KEYWORDS

feature model, product line, survey, evaluation, benchmark

ACM Reference Format:
Chico Sundermann, Vincenzo Francesco Brancaccio, Elias Kuiter, Sebastian Krieter, Tobias Heß, and Thomas Thüm. 2024. Collecting Feature Models from the Literature: A Comprehensive Dataset for Benchmarking. In *28th ACM International Systems and Software Product Line Conference (SPLC '24), September 02–06, 2024, Dommeldange, Luxembourg*. ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3646548.3672590

1 INTRODUCTION

Feature models are commonly used to specify the valid configurations of a product line [6, 8, 51]. Typically, a feature model consists of a set of features and dependencies between those features [51]. A product of the product line is identified by a configuration of features that adheres to all specified dependencies. Manually analyzing feature models is generally infeasible [8, 102]. Even simpler problems, such as checking whether a given configuration satisfies all imposed constraints, require all dependencies of a feature model to be considered. Thus, many automated analyses have been proposed to support various activities in product-line engineering, such as modeling [8], configuring [55, 84], and testing [54, 79].

Many feature-model analyses depend on solving computationally complex problems [106]. In particular, feature models are typically translated to propositional formulas, so their analyses can be reduced to Boolean satisfiability (SAT) [8, 63] or model-counting (#SAT) [36, 56, 107] problems, which are computationally complex [34, 106, 113]. As feature models are often analyzed in interactive settings [1, 9, 70], short response times of reasoning engines (e.g., SAT solvers) are in demand. The efficiency (w.r.t. resource consumption) of tools is particularly relevant when dealing with very complex configurable systems, such as the Linux kernel [66, 109].

Building efficient reasoning engines requires empirical evaluations that are transferable to other instances of the targeted scope. Transferability depends on representative datasets [28, 45], as the performance of reasoning engines highly depends, amongst others, on the analysis and model instance [57, 63, 105]. Without comparisons on feature models that reasonably reflect tool performance in practice, it is difficult for practitioners to select the most promising tool and for tool developers to optimize their algorithms.

Using non-representative datasets for benchmarking performance may lead to empirical evaluations that lack external [117] and ecological [3, 50] validity and cannot be transferred into practice [4, 42] due to several aspects. First, it has been observed that runtimes of reasoning engines, such as SAT solvers, often differ vastly between artificial and real-world formulas [4, 52, 63]. Second, when an empirical evaluation includes only few feature models, it is unclear whether any of its conclusions are transferable to other feature models. This transferability is particularly questionable if the considered feature models are all similar regarding certain characteristics, such as structure (e.g., number of features or number of constraints) and application domain (e.g., automotive or operating system). Third, comparing empirical evaluations is difficult if they use varying datasets, even if those datasets contain multiple real-world instances. Currently, different authors typically use varying datasets for empirical evaluations [10, 14, 38, 54, 56, 81, 89, 97, 105] with only small or even no overlap to models from other work.



This work is licensed under a Creative Commons Attribution International 4.0 License.

SPLC '24, September 02–06, 2024, Dommeldange, Luxembourg

© 2024 Copyright held by the owner/author(s).

ACM ISBN 979-8-4007-0593-9/24/09
https://doi.org/10.1145/3646548.3672590

Identifying relevant feature models for an empirical evaluation is a high-effort task. Publicly available feature models are scattered over various publications [2, 11, 46, 52, 54, 78, 81, 84, 86, 87, 97]. Also, available feature-model repositories (e.g., SPLOT [72]) focus on exchange instead of benchmarking [29, 70, 72] and contain a large share of artificial and small feature models [70, 72], which are less suitable for benchmarking performance [28]. Other available collections are limited to few [52, 76] or even exactly one domain [11, 53, 86, 114].

In this work, we aim to collect feature models used across existing empirical evaluations of feature-model analyses and make them more accessible for researchers and practitioners. To this end, we provide a comprehensive dataset including available feature models from previous publications. We envision that our collection improves the external and ecological validity of future empirical evaluations. Our contributions towards that goal are the following:
Systematic Literature Survey (Section 3). We performed a survey examining overall 4,475 publications to identify work that performs empirical evaluations on feature-model analyses.
Analysis of Common Practices (Section 4). We gather insights on publishing behavior, used feature models, and employed reasoning engines to analyze common practices in feature-model evaluations. Our observations confirm that having an easily accessible, comprehensive collection of feature models yields benefits.
Feature-Model Collection (Section 5.1). We gather feature models from replication packages or similar data repositories provided by identified publications. Overall, our collection contains 2,518 feature models from 13 application domains.
Accessible Repository (Section 5.2). We make each feature model publicly available [21] in (1) the originally published format, (2) the Universal Variability Language (UVL) [103], and (3) as a propositional formula in conjunctive normal form (CNF) expressed in the DIMACS format [90]. For each feature model, we provide various information on its structure, semantics, and origin. For easy usage, our repository comes with several functionalities for extracting partial datasets (e.g., from specific domains, formats, or sizes).
Evaluation Against Requirements (Section 6). We evaluate our dataset regarding its suitability for empirical evaluations considering requirements specified for benchmarks in other domains [45].

2 BACKGROUND

In the following, we give a brief overview on feature models, representations for feature models, and analyses on feature models. For a more detailed context, we refer to other relevant literature for definition of feature models [6, 51], representations [7, 24] and analyses on feature models [8, 69, 106].

2.1 Feature-Model Representations

A feature model specifies the set of valid configurations for a product line [6, 8]. Typically, a feature model $FM = (F, C)$ is defined as a set of features F and constraints C over those features. The constraints implicitly specify the set of valid configurations for the feature model. Various representations for feature models have been proposed, differing mostly in their syntax and visualization [6, 48, 51, 92, 103]. However, some languages also support different language constructs with varying levels of expressiveness [48, 108]. Often, constraints and features are limited to Boolean logic (i.e., features are selected or deselected) [8, 54, 54, 73, 80, 105]. Some feature-model extensions enrich the domain of possible values by supporting numerical features [76, 108] or provide more complex constraints [102, 108].

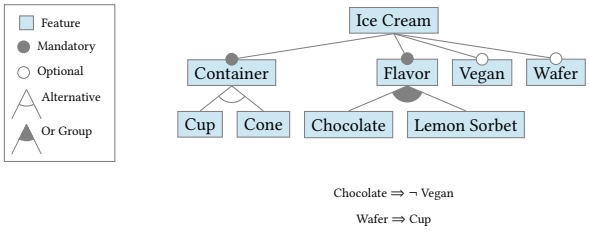


Figure 1: Running Example as a Feature Diagram

Feature Diagrams A feature diagram consists of a hierarchical tree structure over the features and additional cross-tree constraints [92], typically in propositional logic [51, 70, 103]. Figure 1 shows a feature diagram representing a simplified ice cream product line. Here, ice cream requires a `Container` and `Flavor` as denoted by their *mandatory* state. Ice cream may be `Vegan` and may contain a `Wafer` as both are *optional* features. The `Container` can be exactly one of `Cup` and `Cone` as indicated by the *alternative* relationship. Furthermore, the *or* relationship denotes that one or more flavors can be selected. In addition to the tree hierarchy, two cross-tree constraints further limit the set of valid configurations (i.e., ice cream variants). `Chocolate` cannot be selected with `Vegan`. An additional `Wafer` can only be selected with a `Cup`.

Universal Variability Language The Universal Variability Language (UVL) is a textual representation for feature models [103]. Several recent and state-of-the-art tools for feature modeling support UVL [23, 31, 41, 44, 65, 101, 104]. In Listing 1, we show the running example in UVL notation. While the core level of the language supports only propositional constraints [103], recent extensions support more expressive constraints, such as arithmetic expressions over numerical values or non-Boolean features [108].

Conjunctive Normal Form Feature models can be represented as logical formulas. In particular, all feature models with only Boolean variables and constraints can be translated into propositional logic [18]. Furthermore, there have been efforts to also translate more complex feature models (e.g., with numerical features) to propositional logic [76]. Formulas in conjunctive normal form (CNF) are commonly used as input for reasoning engines, such as SAT [12, 73, 75] or #SAT [98, 105, 112] solvers. Hence, feature models are often expressed in CNF to simplify automated reasoning [11, 80]. A CNF is a conjunction of clauses, with each clause being a disjunction of literals [58]. CNFs are typically exchanged in the DIMACS format [90]. In Listing 2, we show an excerpt of a CNF representing our running example in DIMACS format. The line starting with `p cnf` indicates that there are 9 variables (one per feature) and 16 clauses. Each successive line describes a clause. For instance, -2 1 0 describes the clause (¬ `Ice Cream` ∨ `Container`).

Listing 1: UVL

```
features
    "Ice Cream"
        mandatory
            Container
                alternative
                    Cup
                    Cone
            Flavor
                or
                    Chocolate
                    "Lemon Sorbet"
        optional
            Vegan
            Wafer
constraints
    Chocolate => !Vegan
    Wafer => Cup
```

Listing 2: DIMACS

```
c 1 Ice Cream
c 2 Container
...
c 8 Wafer
c 9 Vegan
p cnf 9 16
1 0
-2 1 0
-5 1 0
1 -8 0
...
6 -5 7 0
-6 -9 0
3 -8 0
```

2.2 Feature-Model Analyses

Various analyses for feature models have been proposed [8, 25, 53–55, 69, 77, 79, 97, 107]. These analyses can be employed for the development of configurable systems, to extract metrics (e.g., for economical estimations), or to support users during the configuration process. Many analyses depend on numerous complex computations, such as SAT [8, 54, 73] or #SAT [80, 107] problems.

Anomalies An *anomaly* corresponds to a potentially unintended behavior of the feature model [68, 77]. For example, a contradiction in constraints of the feature model may lead to a *void* feature model [8], from which no valid configuration can be derived. Other anomalies considered in the literature are *dead* features (i.e., features that appear in no valid configuration), which may require solving numerous SAT problems [106].

Sampling With *sampling*, a set of valid configurations following a specific goal is derived. With *t-wise* sampling, a set of valid configurations is created that covers all t-wise interactions [47, 54]. For instance, a 2-wise sample covers all interactions between each pair of features. State-of-the-art t-wise samplers rely on repetitive SAT calls to derive a valid sample [47, 54]. Another popular strategy is *uniform random sampling* [37, 80, 88, 97]. To ensure a uniform distribution, state-of-the-art random samplers typically rely on solving #SAT problems [37, 81, 88, 97].

Feature-Model Counting Many analyses depend on *feature-model counting* (i.e., computing the number of valid configurations for a given feature model) [40, 56, 106, 107]. For example, the number of valid configurations that contain a certain feature can be used to prioritize features during development [107]. Such analyses often rely on numerous #SAT invocations.

3 LITERATURE SURVEY

With our literature survey, we aim to (1) collect a comprehensive dataset of feature models used as test instances in empirical evaluations on feature-model analysis and (2) analyze common practices in evaluations to further inspect the demand for such a collection. In the following, we present our methodology to identify work of interest, collect feature models, and extract relevant information.

3.1 Goals

The main goal of our survey is collecting a feature-model dataset that improves ecological [50] (i.e., transferability to real world) and external validity for benchmarking performance of reasoning engines. Following considerations for selecting benchmark instances from other domains, namely graphs [45], machine learning [61], face recognition [35], and logic [26, 28], we consider three properties relevant for feature models in the context of benchmarking.

P1 *Heterogeneity.* A homogeneous (regarding domain and structure) dataset may limit the transferability [45]. Feature models are used to model variability in various domains, such as automotive [56], system software [11], and finances [27]. As results cannot be necessarily transferred between domains [4, 28, 61], we aim to collect feature models from *many domains*. Also, instances in benchmark datasets should be *heterogeneous considering structural properties* [45], such as number of features or tree hierarchy.

P2 *Suitability for Performance Tests.* There are various aspects of datasets that might be relevant for performance tests depending on the use case. We consider two dimensions here that we found relevant for the evaluation of reasoning engines for own evaluations and for evaluations many we examined during our survey. First, to effectively select and improve algorithms, we need *sufficiently hard* instances (here: feature models) that show the impact of optimizations [28, 45]. Feature models that are easy to analyze with unoptimized algorithms may, thus, be less relevant for a collection that focuses on benchmarking. While the hardness depends on various properties, existing evaluations suggest that feature models with very few features are typically not hard for most feature-model analyses [25, 39, 89, 105]. As feature models with fewer than 100 features appear to be computationally easy for popular automated reasoning engines [89, 102, 105, 106], we only consider feature models with *at least 100 features*. Collecting all feature models available, including very small ones, would considerably increase the required effort and various smaller feature models are already available in the widely used SPLOT repository [72]. We apply this filter of feature models with at least 100 features only for the collection. In the *survey*, we also examine work with smaller feature models. Second, a common goal of performance tests is to *accurately identify performance differences of tools* which requires a *sufficient number* of input instances [35], which are currently laborious to collect.

P3 *Transferability.* The results of evaluations should be transferable into practice [45, 115]. It has been shown that reasoning engines often perform substantially different on artificial instances compared to real-world instances [4, 52, 63]. While empirical evaluations on carefully designed artificial models can still be useful, we only consider *real-world feature models* to ensure ecological validity [50].

In addition to collecting feature models, we aim to use our survey to gather insights on the usage of feature models and reasoning engines in empirical evaluations. We envision that our observations help in understanding the demands and practices in evaluations on feature-model analysis and provide more evidence for problems addressed in this work. Note that while we only consider feature models satisfying **P1**-**P3** for our feature-model collection, we also include empirical evaluations using other feature models in the survey to reduce the bias in our observations on common practices.

Table 1: Search String

	("feature model" OR "feature models" OR "feature modeling" OR "feature modelling")
AND	("product line" OR "product lines" OR "product family")
AND	(analy* OR evaluat* OR solver OR benchmark* OR configura* OR sampl* OR anomal* OR tool*)

Table 2: Inclusion & Exclusion Criteria

IC1	Feature models describe a product line
IC2	Feature models are in a standardized format
IC3	Evaluation with an automated reasoning engine
IC4	Full text of publication is available
IC5	Publication was subject to peer-review
EC1	Publication is not in English
EC2	Paper was published prior to 1990

We provide insights on the considered feature models, the publishing behavior, and evaluated reasoning engines with the following three research questions:

RQ1 How often are feature models reused across evaluations?

RQ2 What kinds of feature models are evaluated?

RQ3 Which reasoning engines are used in empirical evaluations?

3.2 Literature Search

Our literature survey for identifying relevant publications consists of two phases. In the first phase, we perform a keyword search on popular databases to identify publications including relevant empirical evaluations and, thus, feature models. In the second phase, we perform a partial snowballing procedure on the identified papers.

Keyword Search In Table 1, we show the search string we used to identify relevant literature. The search string is inspired by related work also surveying automated analysis on feature models [30, 36]. The search string consists of three main parts aiming to identify work on feature models in the product-line domain that targets a kind of analysis or include an empirical evaluation. Using the search string shown in Table 1, we performed a search on the following literature databases: *ACM Digital Library*, *IEEE Xplore*, *Springer Link*, and *Web of Science*, as we expect a reasonable coverage of relevant publications from these databases.

After collecting the publications matching our keyword search, we apply the inclusion and exclusion criteria shown in Table 2. Here, we aim to only keep publications that cover an empirical evaluation on automated reasoning for feature models in the product line domain. As Kang et al. [51] introduced the notion of feature models in 1990, we exclude earlier publications.

Snowballing After collecting the initial set of publications, we aim to provide additional insights on the included feature models. First, we aim to identify the original publications of feature models we found. Second, we try to collect work that uses feature models originally published in work we identified.

To identify the original publications of found feature models, we perform partial *backward snowballing*. Hereby, we consider feature models included in an empirical evaluation, but originally published in other work. For each publication including such a feature model, we follow the corresponding reference to identify the original work.

Table 3: Data Extraction Form

ID	Data	Purpose
D1	Feature-model publication metric	**RQ1**
D2	Feature-model artificiality metric	**RQ2**
D3	Feature-model size	**RQ2**
D4	Feature-model format	**RQ2**
D5	Feature-model domain	**RQ2**
D6	Solver identification metric	**RQ3**

To identify work that reuses publicly available feature models, we perform *forward snowballing*. Hereby, we consider each work in our literature set that originally published at least one original feature model with 100 or more features for forward snowballing. We extend our literature set with identified publications that include one of those feature models in their empirical evaluation.

For both directions of snowballing, we collect the publications using the *cited by* function from Google Scholar. For each identified paper, we documented each decision for comprehensibility [22].

3.3 Data Extraction

For each identified work satisfying our inclusion and exclusion criteria, we gather insights on the published feature models, how they are published, and with which solvers the feature models were analyzed. In addition to documentary information, such as authors, title, and links, we gathered the information depicted in Table 3. To extract the respective information, the second author searched abstract, introduction, evaluation chapters, and respective artifacts. If a decision about inclusion or the data extraction was unclear, a second author also reviewed the respective work [13].

With the publication metric (D1) shown in Table 4, we examine how the feature models are published. The metric provides insights on the reproducibility and transferability to practice. A score **0** implies that results cannot be reproduced at all. Publications with a score of **1** use generated, artificial models which is out of our scope. We use scores **2** and **3** for forward and backward snowballing (cf. Section 3.2), respectively, to identify additional publications covering real-world feature models.

The artificiality metric (D2) indicates whether a feature model represents a real-world system, is artificial, or of unknown origin. We based the classifications of artificiality on the descriptions in the manuscript. Hereby, we consider the feature model to be real-world if (1) the authors explicitly state that it is a real-world feature model or (2) the descriptions clearly imply that a real-world system is considered. An example for the latter is that the authors state that they used a Linux feature model for their evaluation. If the information does not clearly imply the artificiality, we consider it as *origin unclear*. We later use the metric D2 for identifying feature models satisfying **P3** for our dataset.

With D3-D5, we provide further insights on the feature models by extracting their size (regarding number of features and constraints), original textual format, and domain (e.g., system software or automotive). Note that we select the domain depending on the available information by the following order (1) information by the original authors (2) from other feature models we found with domain information available, or (3) classification of the authors of this work based on content and description.

Table 4: Publication Metric D1

Public. Score	Availability of included feature models
3	Unique feature models publicly available
2	Feature models from other publications
1	Feature models from generator
0	Feature models not published

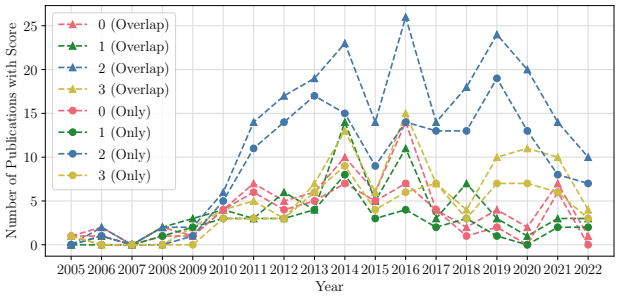


Figure 2: Publication Metric over Time

Orthogonal to analyzing the feature models, we examine the usage of reasoning engines in empirical evaluations. With D6, we analyze the level of information on the reasoning engine provided by authors. We classify the level of detail in: providing a direct link to the used version, specifying the solver name and version, only giving the name of the solver, or not providing sufficient information to identify the solver at all.

4 SURVEY INSIGHTS

With our initial keyword search, we found 3,382 publications in total that conform to our search string. The databases were accessed in May 2022 and, thus, include publications between January 1990 and May 2022. After applying the inclusion and exclusion criteria and filtering duplicates, we identified a set of 345 papers relevant for our survey. With our strategy of forward and backward snowballing (cf. Section 3.2), we further examined 1,093 publications. Here, we identified 53 additional papers, resulting in an overall set of 398 publications. For comprehensibility, sheets that indicate our decisions for each publication regarding inclusion/exclusion criteria and data extraction are publicly available [22].

4.1 RQ1: Publishing of Feature Models

In Figure 2, we show the results for the publication metric of the 398 publications over time. The *(only)*-lines indicate how many publications satisfy exactly that score. The *(overlap)*-lines show how many publications satisfy that score but possibly also others. Overall, 39.8% of the publications contain only feature models from other publications and 56% at least one. 17.1% of the publications include only original feature models and 25.6% at least one. 55 (13.8%) publications published none of the feature models used in their empirical evaluation, further 19.6% do not publish all.

In Figure 3, we show the number of times different publications or repositories have been used in empirical evaluations. Feature

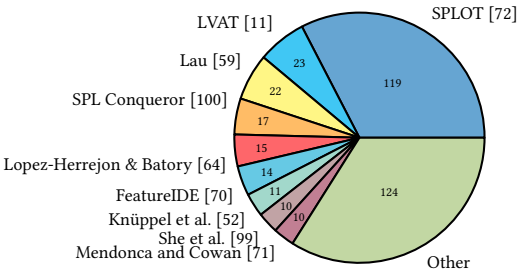


Figure 3: Feature Model Sources

models from the SPLOT [72] repository are by far the most often used with 119 different publications explicitly stating usage of SPLOT models in their empirical evaluation.

RQ1: 19.6% of publications include unpublished feature models in their evaluation. Hence, the feature models cannot be reused in other empirical evaluations and comparisons are more difficult. Results are also not fully reproducible if the feature models are not available. 43.5% of publications do not reuse any previously publicly available feature models.

4.2 RQ2: Feature-Model Characteristics

For 29.4% of the publications, we were not able to identify the origin of all feature models. More than half (53.5%) of the publications include at least one real-world feature model, but also possibly artificial ones. 29.4% of the publications include at least one feature model from generators. Feature models from the following generators are used most often: BeTTy [95] (29 times), the FeatureIDE generator [111] (15 times), and the SPLOT generator [72] (14 times).

We identified a large variety of domains for which feature models are available. However, 102 (25.6%) publications also contain at least one feature model where we were not able to extract the domain. Only 84 (21.1%) of evaluations include feature models from more than three different domains. Feature models from the system software (appearing in 82 publications), e-commerce (80), and automotive (72) domains are included in the largest number of publications. The numbers of occurrences for these domains mainly come from few popular systems, such as Linux for system software, E-Shop [72] for e-commerce, and Automotive02 [52] for automotive, that are included in many publications.

In Figure 4, we show the smallest (x-axis) and largest (y-axis) feature models considering their number of features for each publication. The largest considered feature model includes 1,048,576 features and is artificially generated. The smallest feature model contains only one feature. Overall, we could extract both the minimum and maximum number of features for 324 publications. 100 publications (30.8%) contain only feature models with fewer than 100 features (bottom left square). 233 publications (71.9%) contain at least one feature model with fewer than 100 features (upper squares). 87 publications (26.9%) contain only feature models with at least 100 features (upper right square).

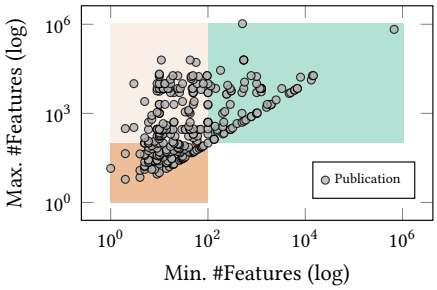


Figure 4: Minimum and Maximum Number of Features

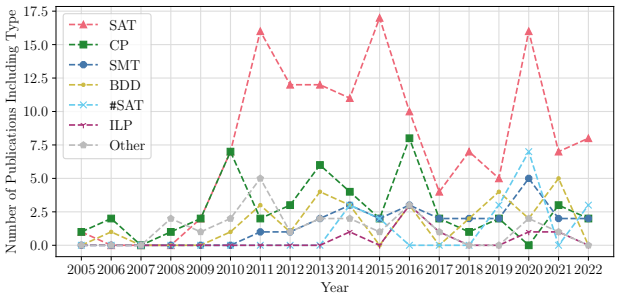


Figure 5: Usage of Solver Classes Over Time

Table 5: Results on Solver Identification

Provided Information	Absolute	Relative
No Information	18	9.7%
Only Name	72	38.7%
Name & Version	22	11.8%
Name & Link	56	30.1%
Name & Version & Link	18	9.7%

RQ2: Real-world feature models seem to be generally more popular than artificial ones for empirical evaluations. However, information on the origin of feature models is often missing, which makes it hard to identify real-world models. While feature models from many domains have been used overall, only 21.1% of the empirical evaluations consider more than three domains. 30.8% of the empirical evaluations are limited to feature models with < 100 features.

4.3 RQ3: Reasoning on Feature Models

In Table 5, we show the type of information documenting the usage of solvers for the 186 publications (46.7%) in our literature set that explicitly report the use of a solver. The remaining publications use ad-hoc solutions, provide no information on a possibly employed reasoning engine, or do not consider automated reasoning at all. The latter were added by the backward snowballing, because they contain models used by included work. Of these 186 publications, 18 (9.7%) do not provide sufficient information to extract even the name of the solver used. Full information (i.e., name, version, and link) is only provided in 18 papers (9.7%).

Figure 5 shows the occurrences of the different solver classes in evaluations over time. SAT solvers are used most in almost every year with overall 149 publications. Solving techniques, such as CP (48), BDDs (28), SMT (27), and #SAT (18), are also often used. The popularity of CP dropped over time, being the second most often used solver 2005–2016. In the last years of the considered time period (2017–2022), CP is only the fifth most popular solver class. 13 of 18 publications considering #SAT solvers were published in the last five years. Overall, the Java-based SAT solver Sat4J [60] is used most (56 publications). The constraint programming (CP) solver Choco [49] and the satisfiability modulo theories (SMT) solver Z3 [19] are used in 30 and 20 publications, respectively. Unidentifiable SAT solvers are also used (17). The most used binary decision diagram (BDD) tool is JavaBDD [116] (13). For model counting (#SAT), sharpSAT [112] is the most often used solver (8).

RQ3: SAT solvers are generally the most popular type of solvers for feature-model analysis with Sat4J [60] being the most used SAT solver. Other popular classes of reasoning engines are CP solvers, BDDs, SMT solvers, and #SAT solvers. 48.4% of publications miss relevant information for reproducibility (e.g., version of solvers).

4.4 Threats to Validity

Validation of Selected Papers The majority of publications in the full dataset (i.e., before applying inclusion and exclusion criteria) were only evaluated by one author. To reduce the effect, each time this author was not fully certain on whether a publication should be included, another author also checked the paper. All decisions and reasoning behind them, in terms of fulfilled or violated inclusion/exclusion criteria, are publicly available [22].

Dependency on Other Author's Information For identifying properties of the feature model (e.g., the artificiality), we rely on the information provided by the authors. We tried to verify some properties, such as domain and artificiality, from provided artifacts. However, as of November 2022, the feature models for many publications are not publicly available (anymore).

Interpretation of Authors In several cases, the classification of feature models, regarding their artificiality and their domain, required interpretation from the authors of this work. Hence, some classifications possibly would have been decided differently by other authors. To reduce the impact of our decisions, we tried to base the decision as much as possible on information provided by the publishers of the feature models. Further, each unclear feature model was examined by at least two authors.

Limited Scope Our literature survey focuses on feature models that have previously been used in evaluations. As a consequence, we might have missed work that published relevant feature models but does not evaluate them for any kind of feature-model analysis. We still expect that the feature models we collected are particularly relevant for better comparability to previous empirical evaluations. Also, as we include feature models from many publications with differing domains and structures, we expect that our dataset is beneficial for practitioners and researchers with regard to the properties outlined in Section 3.1. Nevertheless, we consider extending our collection in the future as valuable.

5 FEATURE-MODEL DATASET

The main goal of this work is to derive a feature-model collection that can be used for comparable empirical evaluations that can be transferred into practice. We limit our dataset to feature models that represent a real-world system (cf. **P3** in Section 3.1) and have at least 100 features (cf. **P2**). Overall, we found 2,518 feature models that match our criteria. In the following, we give an overview on the feature models in our collection and the functionalities of the public repository we provide [21].

5.1 Feature Models

To identify the feature models we included in our repository, we further filtered the publications according to the properties **P1**-**P3** defined in Section 3.1. After filtering, we identified 20 papers with relevant, original, and publicly available feature models resulting in overall 2,518 feature models from 41 different systems (e.g., Linux) and 13 application domains (e.g., system software). Table 6 gives an overview on the collected feature models. Since multiple system software feature models were only available in CNF format, we show the number of clauses for better comparability between the formats. Also, for feature models only available in CNF, we consider the number of variables as number of features in Table 6.

While we identified 102 papers with original feature models, we excluded the majority of those, as they only included small or artificial original models or their repositories were not available anymore. We also collected seven feature-model histories, in which each feature model represents an evolution step of the corresponding system. In some cases, the feature model did not change after an evolution step [85, 86], here we discarded all duplicates but the first occurring. Without counting each feature model of histories as different instances, our collection contains 1,617 feature models. The high number of security feature models comes from the AMADEUS-dataset [114] where each feature model describes configurations that might be affected by a cybersecurity vulnerability.

5.2 Repository

Collection of Feature Models We provide each identified feature model in the format of the original publication, in UVL [103], and CNF (DIMACS).[1] We added UVL to facilitate standardized feature-model analyses, which may also rely on the feature hierarchy of the model. The DIMACS format allows users the quick use of off-the-shelf solvers, such as SAT [12, 73] or #SAT [105, 112] solvers. If a feature model's history was made publicly available, we provide all feature models representing the history.

To enable translations from every identified format, we use the transformation approach TraVarT [23], with UVL acting as pivot language (i.e., intermediate language for transformations). The most recent version of TraVarT[2] was lacking support for FeatureIDE [70], AFM [9], DIMACS, and SXFM [72], so we extended TraVarT with respective transformations.[3] For Clafer [48], we could not identify a transformation to one of the supported formats. However, every feature model we found in Clafer was also available in a format supported by TraVarT [11, 52], and we used those instead. For translation to CNF, our TraVarT extension internally uses a distributive transformation implemented in FeatureIDE [70] (v3.9.2), which preserves logical equivalence and introduces no artificial variables [58]. As some formats have different levels of expressiveness, the conversions to UVL or CNF may result in information loss [24]. In particular, our collection includes feature diagrams as presented in Section 2.1, extended feature models which contain non-functional attributes [8], and cardinality-based feature models [17]. However, in our survey we did not identify a single feature model that included constructs that cannot be semantically equivalently (i.e., represents same configuration space) represented in CNF or UVL.[4] The UVL parser [108] (also used in TraVarT) supports semantic equivalence preserving conversions to Boolean logic, which enables converting more complex expressions into CNF. In some cases, SXFM models included the same feature name twice which is not allowed in UVL [72]. Here, we resolve the duplicate feature by introducing a new feature and a respective cross-tree constraint (i.e., for duplicate feature f rename duplicate to f' with $f \iff f'$). Also, CNF does not support storing attributes, so they are discarded. In both cases, we have syntactical loss (i.e., same configuration space but different representation) [24].

Meta-Data on Feature Models For each feature model included in our repository, we provide a variety of different metrics and general information. Table 7 shows an excerpt of the different types of metrics we include in the repository separated into three categories. First, we provide structural metrics, such as the number of features. Second, we include metrics which require some semantic analyses of the feature model (e.g., number of valid configurations). We compute the structural and semantic metrics with a script based on the FeatureIDE library [70].[5] Third, we present information on the origin of the feature model, such as references (DOI) to the original work or the tooling used for conversion.

Dataset Extraction As the repository already contains 2,518 feature models today, using all of them for an empirical evaluation may be too costly. An analysis may also be only relevant for certain domains (e.g., software). In this case, users of our feature-model dataset may want to use only a subset of the feature models. To ease extracting datasets depending on the use case, we provide a Python-based script that supports to filter feature models based on the provided meta-data. Users can filter based on various properties, such as name, tags, structure, domain, or whether they are part of a history. For instance, a user of the repository may derive all feature models with more than *1,000 features* from the *software* or *automotive* domain. The extractor then provides a separate instance with the feature models matching the criteria, statistics over this dataset, and a configuration file for reusing the exact same dataset.

Dataset Configurations The extractor allows the user to derive a feature-model dataset based on configuration files. Mainly, those configurations are used to describe datasets employed in other empirical evaluations to enable easier comparability. The configurations include information on applied filters (e.g., domain), list

[1] Few feature models violated the respective language specification and could not be fully translated to UVL and DIMACS.
[2] https://github.com/SECPS/TraVarT
[3] https://zenodo.org/doi/10.5281/zenodo.11654485
[4] Note that the feature models we found may be already simplified. For instance, tri-state features from KConfig [15] may be simplified to Boolean features [82].
[5] https://zenodo.org/doi/10.5281/zenodo.11653333

Table 6: Feature Models Included in Our Repository

Domain	Feature Models	Systems	Histories	Features	Clauses	Origins
Automotive	5	2	1	2,513–18,253	666–2,833	[52, 53]
Business Software	1	1	0	1,920	59,044	[84]
Cloud	1	1	0	131	14	[33]
Database	1	1	0	117	282	[43]
Deep Learning	2	1	0	3,296–6,867	9–76	[32]
E-Commerce	2	2	0	17–2,238	0	[59, 84]
Finance	13	4	1	142–774	4–1,148	[2, 72, 77]
Games	1	1	0	144	0	[94]
Hardware	2	2	0	172–364	0–12	[72, 102]
Navigation	2	2	0	103–145	2–13	[72]
Security	1,464	2	0	101–4,351	1–8,138	[72, 114]
System Software	1,025	21	5	179–80,258	26–767,040	[2, 11, 52, 72, 78, 81, 83, 85–87]
DSL Development	1	1	0	137	1	[72]
Overall	**2,518**	**41**	**7**	**101–80,258**	**0–767,040**	

Table 7: Excerpt of Provided Metrics

Structure	Semantic†	Origin
#Features	#Valid Configurations	Publication (DOI)
#Constraints	Void	Source (URL)
Tree Depth	#Dead Features	Conversion Tool
#Leaf Features	#Atomic Sets	Domain
Constraints Density	#Core Features	Keywords

† For few feature models, metrics are missing due to scalability issues.

of used feature models, and information on the performed experiments. For now, we include some manually created configurations that describe datasets in existing empirical evaluations. We created configurations for different types of analyses, namely t-wise sampling [54], uniform random sampling [81], validity checks [63], and feature-model counting [105]. We envision that future publications can publish their configuration and reference it.[6]

Dataset Maintenance In the future, we aim to maintain and update the collection regarding the following aspects. First, we ensure that the collection is long-term publicly available. Second, we continue to add feature models that meet our criteria. Third, we review proposals for new additions. Fourth, we update the metadata about the models after new additions.

6 EVALUATING AGAINST REQUIREMENTS

In this section, we aim to provide insights on the suitability of our dataset for empirical evaluations on performance of feature-model analyses. In particular, we examine different indicators on how well our dataset satisfies our requirements **P1**–**P3** which we adopted from benchmark datasets from other domains (cf. Section 3.1).

6.1 P1: Heterogeneity

In this section, we examine the heterogeneity of our dataset regarding structural properties of the feature models. Figure 6 shows the following metrics for each feature model: the number of features, number of clauses, the connectivity[7], share of features appearing in cross-tree constraints (ECR [74]), share of features that are optional (i.e., neither core nor dead), and the number of configurations. The colors of the markers indicate the domain of the feature model. Overall, the collection covers a wide range for every metric with, for instance, 101–800,258 features, 0–767,040 clauses, 10^1–10^{1534} valid configurations. Checking the similarity over various metrics, feature models from different systems structurally differ in most cases. However, feature models from similar systems are often more similar regarding structural metrics. In particular, feature models within the same history from the systems FinancialServices, uClibc, BusyBox, Fiasco, and Linux, respectively, are very similar based on various extracted metrics. Those similarities are already observable on the example metrics shown in Figure 6, as the histories form clusters in the diagrams. For instance, the standard deviation of the number of features over all feature models normalized by the mean is 4.5, while in a given history it is at most 0.32. Similarly, for constraints the overall value is 5.4 and for the different histories at most 0.44. Furthermore, the CDL feature models [52], which are not a history but variants with a shared core, are also highly similar.

P1: Overall, our dataset covers a wide range of structural properties and most models are considerably different regarding those properties. Still, there are some clusters of feature models with very similar structural properties. Hence, it may yield benefits to only use some candidates of those similar models for evaluations.

6.2 P2: Benchmark Suitability

Here, we examine the suitability of our dataset for benchmarking regarding the two aspects for **P2** introduced in Section 3.1. First, we check if there are hard (i.e., induce long runtimes for some solvers) instances that can be used for effectively improving existing algorithms. Second, we evaluate if our dataset is sensitive enough to detect significant differences between the performances of different tools. To this end, we measure runtimes of differently performing tools, apply common statistical tests, and examine the resulting p-values. Since many feature model analyses rely on SAT and #SAT solving [106], we analyze solvers of both categories. In particular,

[6] Example dataset from Liang et al. [63]: https://github.com/SoftVarE-Group/feature-model-benchmark/blob/master/paper_configs/Liang2015.json

[7] The connectivity describes the average number of literals a feature is connected to via constraints.

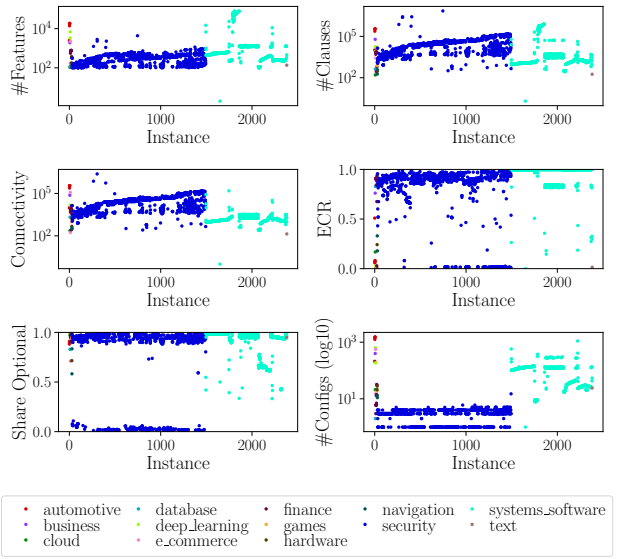


Figure 6: Structural Properties of Dataset

for SAT, we use `Kissat` and `SBVA-Cadical` which performed best at the latest (2023) SAT competition [28] and `Glucose` which is another popular SAT solver employed in Sat4J [60] and in relevant variability modelling tools [31, 70]. For #SAT, we use the three best performing #SAT solvers from the model counting competition 2023.[8] The replication package for the runtime measurements is publicly available.[9]

Figure 7 shows the median[10] runtimes on our feature-model dataset of the SAT solvers in the first row and of the #SAT solvers in the second row. `Kissat` is the only solver which did not hit a single timeout. Hence, for all solvers but `Kissat` there is considerable potential for further optimizations and instances to evaluate potential improvements. Even for `Kissat`, there are several instances with between 0.1s and 1s of runtime which could still be used for tweaking its performance. As many feature-model analyses rely on numerous SAT invocations [106], even the short runtimes for a single `Kissat` invocation can lead to long response times.

To check the significance of differences between tools, we applied a Friedman test with a post-hoc Conover [16] respectively for both types of solvers. For each pair, we found clearly significant differences with p-values smaller than 10^{-120}. We conclude that the dataset enables significant runtime comparisons even for only smaller differences. For instance, the close difference between overall runtimes of `d4` (96.3 minutes) and `Ganak` (103 minutes) is still clearly significant ($p < 10^{-193}$).

P2: Our dataset appears to satisfy both specified requirements for being suitable for benchmarking. First, there are several hard instances that can be used for further optimization. Second, the dataset can be used to identify differences between solvers with high statistical power (i.e., very small p-values).

[8] https://mccompetition.org/assets/files/2023/MC2023_awards.pdf
[9] https://zenodo.org/doi/10.5281/zenodo.11654446
[10] We performed three repetitions per instance for each solver.

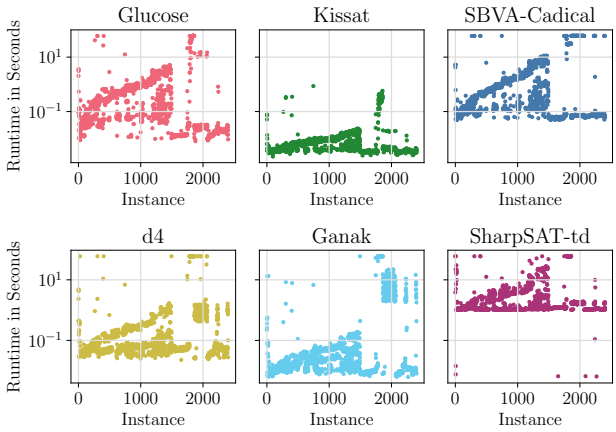


Figure 7: Runtime of Algorithms for SAT and #SAT

6.3 P3: Transferability

We only considered real-world feature models for our dataset. As described in Section 3.3, we chose a rather conservative approach where we only included feature models which we could clearly identify as real-world models or the authors explicitly declared as real-world. Furthermore, our dataset is a superset of publicly available datasets from other empirical evaluations. As we found in our survey, our resulting collection contains more large, real-world feature models and more domains compared to datasets used in previous evaluations, which increases the probability of results being transferable. In addition, the previous usage in automated analyses is an indicator that the feature models in our collection may be relevant instances for future evaluations.

P3: There are several indicators that using feature models in our dataset may improve transferability compared to previously used datasets. However, the representativeness of our dataset for other feature models in practice requires further examination. Still, since we have a superset of feature models used in previous evaluations, we assume a higher coverage of instances relevant in practice compared to previously used datasets. Thus, we expect a practical impact of our collection on future evaluations.

6.4 Discussion

Overall, we found several indicators regarding our specified requirements **P1**–**P3** that our dataset can be beneficial for empirical evaluations. First, the dataset is a superset of existing datasets used in existing empirical evaluations and covers a wide range of structural properties and domains. Second, the included feature models enable comparisons of the performance of different tools with high statistical significance. Third, due to our search criteria, we only included real-world feature models that were already used in other empirical evaluations. Still, some insights on the heterogeneity suggest that using a subset of the collection may be beneficial. We found some clusters of variants and versions of feature models that are structurally similar. As a consequence, we provide filters that support extracting collections of only one feature model per version/variant of the same cluster.

7 RELATED WORK

Ecological Validity for Feature-Model Analysis Ecological validity describes the transferability of studies or evaluations to the real world [50], which is a major motivation for our work. Kamali et al. [50] also target ecological validity by analyzing the usage of variability models in the literature. They use the literature set collected by Galindo et al. [30] who targeted feature-model analysis. Hence, their and our survey have a similar scope. However, Kamali et al. [50] limit their elaborations on explicit open-source projects. As a consequence, they identify around[11] 80 feature models with more than 100 features, while we identified 1,617 models, not counting histories. In contrast to our work, Kamali et al. [50] also do not provide an accumulated collection of feature models.

Feature-Model Collections Several repositories providing a collection of feature models are available [11, 29, 70, 72, 91, 103]. According to the insights from our literature survey, feature models from the SPLOT [72] repository are most commonly used. Still, SPLOT contains rather small real-world feature models with 98.3% (17 out of 997) having less than 100 features. The largest real-world SPLOT feature model contains 451 features. In contrast to our collection, none of the available repositories target datasets that reasonably reflect performance in practice, but target the exchange of all kinds of feature models, including academic models.

In several publications, the authors published considerable collections including real-world feature models [5, 11, 52, 53, 76, 86, 114]. However, these datasets are limited to only few [52, 76] or even exactly one domain [11, 53, 86, 114]. As our collection is a superset of existing datasets, we cover various domains (13) and feature model sizes (101-80,528 features). Additionally, our collection comes with tool support to simplify extracting a dataset tailored to the requirements of an empirical evaluation.

Benchmarking Efforts Several publications consider benchmarking in the product-line domain [20, 95]. However, neither of the available benchmarks targets real-world feature models from multiple domains. The benchmarks either (1) use generated feature models [95], (2) focus on entirely different scopes [20, 67, 93, 118], and/or (3) are limited to a single domain or even system [67, 118].

Surveys on Feature-Model Analysis Several other surveys target work on feature-model analyses [8, 30, 36, 62, 96, 107, 110]. These surveys focus on the analyses and provide, amongst others, formal specifications [8, 36, 107], possible algorithms [36, 107] or categorizations of analyses [62, 96, 107, 110]. In contrast, we focus on the feature models by identifying properties of the feature model, examining the reuse of feature models across empirical evaluations, and gathering the feature models for our collection.

Benchmarks in Other Domains Automated reasoning engines are typically evaluated on collections of logical formulas [12, 26, 28, 112]. In particular, the SAT competition [28] and the model-counting competition [26] provide large datasets to compare solvers. These collections consist of randomly generated instances and SAT encodings of popular complex problems, such as graph coloring [26, 28]. However, these collections do not include instances representing feature models or configurable systems. Our work could be a stepping stone towards including feature-model instances in future contests. Collections for benchmarking are also used in other domains, such as graphs [45], machine learning [61], and face recognition [35]. While the respective instances are not closely related to feature models, we examined their work to derive benchmarking requirements also relevant for our scope (cf. Section 3.1).

[11]The number is not exact, as information for few feature models is missing.

8 CONCLUSION

Identifying relevant feature models for benchmarking is tedious as they are scattered across numerous publications. Empirical evaluations often include only small feature models and are typically limited to few domains. Also, the sets of feature models used in different evaluations often do not intersect much, which limits external validity and hinders comparability. In this work, we perform a literature survey to gather a collection of real-world feature models for benchmarking. Our dataset contains overall 2,518 real-world feature models with at least 100 features each, made available in the original format, UVL, and DIMACS. We also provide various insights on structure, semantics, origin, and suitability for benchmarking of these feature models. Our observations on the latter suggest that our dataset may be valuable for future evaluations.

In the future, multiple authors plan to maintain the dataset and extend it with new feature models. We also invite researchers and practitioners to contribute models. Extending the repository with data beyond the feature model (e.g., source code) may also be valuable for related analyses. To simplify access, we aim to add our dataset to an existing repository (e.g., UVLHub [91]). Furthermore, we plan on extending recommendations for feature models to include for specific use-cases based on model quality, general user ratings, heterogeneity, and domain.

ACKNOWLEDGMENTS

This work is based on the master's thesis of Brancaccio [13].

REFERENCES

[1] Mathieu Acher, Philippe Collet, Philippe Lahire, and Robert B. France. 2013. Familiar: A Domain-Specific Language for Large Scale Management of Feature Models. *SCP* 78, 6 (2013), 657–681.
[2] Mustafa Al-Hajjaji, Thomas Thüm, Malte Lochau, Jens Meinicke, and Gunter Saake. 2019. Effective Product-Line Testing Using Similarity-Based Product Prioritization. *SoSyM* 18, 1 (2019), 499–521.
[3] Chittaranjan Andrade. 2018. Internal, External, and Ecological Validity in Research Design, Conduct, and Evaluation. *Indian Journal of Psychological Medicine* 40, 5 (2018), 498–499.
[4] Carlos Ansótegui, María Luisa Bonet, and Jordi Levy. 2009. On the Structure of Industrial SAT Instances. In *CP*. Springer, 127–141.
[5] Eduard Baranov, Axel Legay, and Kuldeep S. Meel. 2020. Baital: An Adaptive Weighted Sampling Approach for Improved t-Wise Coverage. In *ESEC/FSE*. ACM, 1114–1126.
[6] Don Batory. 2005. Feature Models, Grammars, and Propositional Formulas. In *SPLC*. Springer, 7–20.
[7] Maurice H. ter Beek, Klaus Schmid, and Holger Eichelberger. 2019. Textual Variability Modeling Languages: An Overview and Considerations. In *SPLC*. ACM, 151–157.
[8] David Benavides, Sergio Segura, and Antonio Ruiz-Cortés. 2010. Automated Analysis of Feature Models 20 Years Later: A Literature Review. *Information Systems* 35, 6 (2010), 615–708.
[9] David Benavides, Sergio Segura, Pablo Trinidad, and Antonio Ruiz-Cortés. 2007. FAMA: Tooling a Framework for the Automated Analysis of Feature Models. In *VaMoS*. Technical Report 2007-01, Lero, 129–134.
[10] David Benavides, Pablo Trinidad, and Antonio Ruiz-Cortés. 2005. Using Constraint Programming to Reason on Feature Models. In *SEKE*. 677–682.
[11] Thorsten Berger, Steven She, Rafael Lotufo, Andrzej Wąsowski, and Krzysztof Czarnecki. 2013. A Study of Variability Models and Languages in the Systems Software Domain. *TSE* 39, 12 (2013), 1611–1640.

[12] Armin Biere. 2008. PicoSAT Essentials. *JSAT* 4 (2008), 75–97.

[13] Vincenzo Brancaccio. 2022. *A Systematic Literature Review Towards a Representative Feature-Model Benchmark.* Master's Thesis. University of Ulm.

[14] Sheng Chen and Martin Erwig. 2011. Optimizing the Product Derivation Process. In *SPLC*. IEEE, 35–44.

[15] The Kernel Development Community. 2018. KConfig Language. Website: https://www.kernel.org/doc/html/latest/kbuild/kconfig-language.html. Accessed: 2024-01-30.

[16] William J Conover and Ronald L Iman. 1981. Rank Transformations as a Bridge Between Parametric and Nonparametric Statistics. *The American Statistician* 35, 3 (1981), 124–129.

[17] Krzysztof Czarnecki and Chang Hwan Peter Kim. 2005. Cardinality-Based Feature Modeling and Constraints: A Progress Report. In *SF*. 16–20.

[18] Krzysztof Czarnecki and Andrzej Wąsowski. 2007. Feature Diagrams and Logics: There and Back Again. In *SPLC*. IEEE, 23–34.

[19] Leonardo De Moura and Nikolaj Bjørner. 2008. Z3: An Efficient SMT Solver. In *TACAS*. Springer, 337–340.

[20] Sebastian Erdweg, Tijs van der Storm, Markus Völter, Laurence Tratt, Remi Bosman, William R. Cook, Albert Gerritsen, Angelo Hulshout, Steven Kelly, Alex Loh, Gabriël Konat, Pedro J. Molina, Martin Palatnik, Risto Pohjonen, Eugen Schindler, Klemens Schindler, Riccardo Solmi, Vlad Vergu, Eelco Visser, Kevin van der Vlist, Guido Wachsmuth, and Jimi van der Woning. 2015. Evaluating and Comparing Language Workbenches: Existing Results and Benchmarks for the Future. *Comput. Lang. Syst. Struct.* 44 (2015), 24–47.

[21] Sundermann et al. 2024. Repository for Feature-Model Dataset. https://zenodo.org/doi/10.5281/zenodo.11652924.

[22] Sundermann et al. 2024. Reproducibility Package for Literature Survey. https://zenodo.org/doi/10.5281/zenodo.11653969.

[23] Kevin Feichtinger, Johann Stöbich, Dario Romano, and Rick Rabiser. 2021. TRAVART: An Approach for Transforming Variability Models. In *VaMoS*. ACM, Article 8, 10 pages.

[24] Kevin Feichtinger, Chico Sundermann, Thomas Thüm, and Rick Rabiser. 2022. It's Your Loss: Classifying Information Loss During Variability Model Roundtrip Transformations. In *SPLC*. ACM, 67–78.

[25] David Fernández-Amorós, Ruben Heradio, José Antonio Cerrada, and Carlos Cerrada. 2014. A Scalable Approach to Exact Model and Commonality Counting for Extended Feature Models. *TSE* 40, 9 (2014), 895–910.

[26] Johannes K. Fichte, Markus Hecher, and Florim Hamiti. 2021. The Model Counting Competition 2020. *JEA* 26, Article 13 (2021), 26 pages.

[27] Claudia Fritsch, Richard Abt, and Burkhardt Renz. 2020. The Benefits of a Feature Model in Banking. In *SPLC*. ACM, Article 9, 11 pages.

[28] Nils Froleyks, Marijn Heule, Markus Iser, Matti Järvisalo, and Martin Suda. 2021. SAT Competition 2020. *AIJ* 301 (2021), 103572.

[29] José A. Galindo and David Benavides. 2019. Towards a New Repository for Feature Model Exchange. In *MODEVAR*. ACM, 170–173.

[30] José A. Galindo, David Benavides, Pablo Trinidad, Antonio-Manuel Gutiérrez-Fernández, and Antonio Ruiz-Cortés. 2019. Automated Analysis of Feature Models: Quo Vadis? *Computing* 101, 5 (2019), 387–433.

[31] José A. Galindo, José Miguel Horcas, Alexander Felfernig, David Fernández-Amorós, and David Benavides. 2023. FLAMA: A Collaborative Effort to Build a New Framework for the Automated Analysis of Feature Models. In *SPLC*. ACM, 16–19.

[32] Salah Ghamizi, Maxime Cordy, Mike Papadakis, and Yves Le Traon. 2019. Automated Search for Configurations of Convolutional Neural Network Architectures. In *SPLC*. ACM, 119–130.

[33] Oscar González-Rojas and Juan Tafurth. 2019. Multi-cloud Services Configuration Based on Risk Optimization. In *On the Move to Meaningful Internet Systems: OTM 2019 Conferences*, Hervé Panetto, Christophe Debruyne, Martin Hepp, Dave Lewis, Claudio Agostino Ardagna, and Robert Meersman (Eds.). Springer, 733–749.

[34] Jun Gu, Paul W Purdom, John Franco, and Benjamin W Wah. 1996. *Algorithms for the Satisfiability (SAT) Problem: A Survey.* Technical Report. Cincinnati University.

[35] Yandong Guo, Lei Zhang, Yuxiao Hu, Xiaodong He, and Jianfeng Gao. 2016. MS-Celeb-1M: A Dataset and Benchmark for Large-Scale Face Recognition. In *Computer Vision – ECCV 2016*. Springer, 87–102.

[36] Ruben Heradio, David Fernández-Amorós, José Antonio Cerrada, and Ismael Abad. 2013. A Literature Review on Feature Diagram Product Counting and Its Usage in Software Product Line Economic Models. *IJSEKE* 23, 08 (2013), 1177–1204.

[37] Ruben Heradio, David Fernández-Amorós, José Antonio Galindo, and David Benavides. 2020. Uniform and Scalable SAT-Sampling for Configurable Systems. In *SPLC*. ACM, 17:1–17:11.

[38] Ruben Heradio, David Fernández-Amorós, Christoph Mayr-Dorn, and Alexander Egyed. 2019. Supporting the Statistical Analysis of Variability Models. In *ICSE*. IEEE, 843–853.

[39] Ruben Heradio, Héctor José Pérez-Morago, David Fernández-Amorós, Roberto Bean, Francisco Javier Cabrerizo, Carlos Cerrada, and Enrique Herrera-Viedma. 2016. Binary Decision Diagram Algorithms to Perform Hard Analysis Operations on Variability Models. In *SOMET*. IOS Press, 139–154.

[40] Rubén Heradio-Gil, David Fernández-Amorós, José Antonio Cerrada, and Carlos Cerrada. 2011. Supporting Commonality-Based Analysis of Software Product Lines. *IET Software* 5, 6 (2011), 496–509.

[41] Tobias Heß, Tobias Müller, Chico Sundermann, and Thomas Thüm. 2022. ddueruem: A Wrapper for Feature-Model Analysis Tools. In *SPLC*. ACM, 54–57.

[42] Tobias Heß, Chico Sundermann, and Thomas Thüm. 2021. On the Scalability of Building Binary Decision Diagrams for Current Feature Models. In *SPLC*. ACM, 131–135.

[43] Robert M. Hierons, Miqing Li, Xiaohui Liu, Jose Antonio Parejo, Sergio Segura, and Xin Yao. 2020. Many-Objective Test Suite Generation for Software Product Lines. *TOSEM* 29, 1, Article 2 (2020), 46 pages.

[44] Jose M. Horcas, Jose A. Galindo, Mónica Pinto, Lidia Fuentes, and David Benavides. 2022. FM Fact Label: A Configurable and Interactive Visualization of Feature Model Characterizations. In *MODEVAR (SPLC)*. ACM, 42–45.

[45] Weihua Hu, Matthias Fey, Marinka Zitnik, Yuxiao Dong, Hongyu Ren, Bowen Liu, Michele Catasta, and Jure Leskovec. 2020. Open Graph Benchmark: Datasets for Machine Learning on Graphs. *Advances in Neural Information Processing Systems (NeurIPS)* 33 (2020), 22118–22133.

[46] Martin Fagereng Johansen, Øystein Haugen, and Franck Fleurey. 2011. Properties of Realistic Feature Models Make Combinatorial Testing of Product Lines Feasible. In *MODELS*. Springer, 638–652.

[47] Martin Fagereng Johansen, Øystein Haugen, and Franck Fleurey. 2012. An Algorithm for Generating T-Wise Covering Arrays From Large Feature Models. In *SPLC*. ACM, 46–55.

[48] Paulius Juodisius, Atrisha Sarkar, Raghava Rao Mukkamala, Michal Antkiewicz, Krzysztof Czarnecki, and Andrzej Wasowski. 2018. Clafer: Lightweight Modeling of Structure, Behaviour, and Variability. *CoRR* (2018).

[49] Narendra Jussien, Guillaume Rochart, and Xavier Lorca. 2008. Choco: An Open Source Java Constraint Programming Library. In *OSSICP*. CCSD-HAL.

[50] Seiede Reyhane Kamali, Shirin Kasaei, and Roberto E. Lopez-Herrejon. 2019. Answering the Call of the Wild? Thoughts on the Elusive Quest for Ecological Validity in Variability Modeling. In *MODEVAR*. ACM, 143–150.

[51] Kyo C. Kang, Sholom G. Cohen, James A. Hess, William E. Novak, and A. Spencer Peterson. 1990. *Feature-Oriented Domain Analysis (FODA) Feasibility Study.* Technical Report CMU/SEI-90-TR-21. Software Engineering Institute.

[52] Alexander Knüppel, Thomas Thüm, Stephan Mennicke, Jens Meinicke, and Ina Schaefer. 2017. Is There a Mismatch Between Real-World Feature Models and Product-Line Research?. In *ESEC/FSE*. ACM, 291–302.

[53] Matthias Kowal, Sofia Ananieva, and Thomas Thüm. 2016. Explaining Anomalies in Feature Models. In *GPCE*. ACM, 132–143.

[54] Sebastian Krieter, Thomas Thüm, Sandro Schulze, Gunter Saake, and Thomas Leich. 2020. YASA: Yet Another Sampling Algorithm. In *VaMoS*. ACM, Article 4, 10 pages.

[55] Sebastian Krieter, Thomas Thüm, Sandro Schulze, Reimar Schröter, and Gunter Saake. 2018. Propagating Configuration Decisions With Modal Implication Graphs. In *ICSE*. ACM, 898–909.

[56] Andreas Kübler, Christoph Zengler, and Wolfgang Küchlin. 2010. Model Counting in Product Configuration. In *LoCoCo*. Open Publishing Association, 44–53.

[57] Elias Kuiter, Tobias Heß, Chico Sundermann, Sebastian Krieter, Thomas Thüm, and Gunter Saake. 2024. How Easy Is SAT-Based Analysis of a Feature Model?. In *VaMoS*. ACM, 149–151.

[58] Elias Kuiter, Sebastian Krieter, Chico Sundermann, Thomas Thüm, and Gunter Saake. 2022. Tseitin or not Tseitin? The Impact of CNF Transformations on Feature-Model Analyses. In *ASE*. ACM, 110:1–110:13.

[59] Sean Quan Lau. 2006. *Domain Analysis of E-Commerce Systems Using Feature-Based Model Templates.* Master's thesis. University of Waterloo.

[60] Daniel Le Berre and Anne Parrain. 2010. The Sat4j Library, Release 2.2. *JSAT* 7, 2-3 (2010), 59–64.

[61] Tai Le Quy, Arjun Roy, Vasileios Iosifidis, Wenbin Zhang, and Eirini Ntoutsi. 2022. A Survey On Datasets For Fairness-Aware Machine Learning. *WIREs Data Mining and Knowledge Discovery* 12, 3 (2022).

[62] Jihyun Lee, Sungwon Kang, and Danhyung Lee. 2012. A Survey on Software Product Line Testing. In *SPLC*. ACM, 31–40.

[63] Jia Hui Liang, Vijay Ganesh, Krzysztof Czarnecki, and Venkatesh Raman. 2015. SAT-Based Analysis of Large Real-World Feature Models Is Easy. In *SPLC*. Springer, 91–100.

[64] Roberto E. Lopez-Herrejon and Don Batory. 2001. A Standard Problem for Evaluating Product-Line Methodologies. In *GCSE*. Springer, 10–24.

[65] Jacob Loth, Chico Sundermann, Tobias Schrull, Thilo Brugger, Felix Rieg, and Thomas Thüm. 2023. UVLS: A Language Server Protocol for UVL. In *SPLC*. ACM, 43–46.

[66] Rafael Lotufo, Steven She, Thorsten Berger, Krzysztof Czarnecki, and Andrzej Wąsowski. 2010. Evolution of the Linux Kernel Variability Model. In *SPLC*. Springer, 136–150.

[67] Jabier Martinez, Tewfik Ziadi, Mike Papadakis, Tegawendé F. Bissyandé, Jacques Klein, and Yves Le Traon. 2016. Feature Location Benchmark for Software

Families Using Eclipse Community Releases. In *ICSR*. Springer, 267–283.
[68] Jacopo Mauro, Michael Nieke, Christoph Seidl, and Ingrid Chieh Yu. 2017. Anomaly Detection and Explanation in Context-Aware Software Product Lines. In *SPLC*. ACM, 18–21.
[69] Flávio Medeiros, Christian Kästner, Márcio Ribeiro, Rohit Gheyi, and Sven Apel. 2016. A Comparison of 10 Sampling Algorithms for Configurable Systems. In *ICSE*. ACM, 643–654.
[70] Jens Meinicke, Thomas Thüm, Reimar Schröter, Fabian Benduhn, Thomas Leich, and Gunter Saake. 2017. *Mastering Software Variability With FeatureIDE*. Springer.
[71] Marcilio Mendonca and Donald Cowan. 2010. Decision-Making Coordination and Efficient Reasoning Techniques for Feature-Based Configuration. *SCP* 75, 5 (2010), 311–332.
[72] Marcílio Mendonça, Moises Branco, and Donald Cowan. 2009. S.P.L.O.T.: Software Product Lines Online Tools. In *OOPSLA*. ACM, 761–762.
[73] Marcílio Mendonça, Andrzej Wąsowski, and Krzysztof Czarnecki. 2009. SAT-Based Analysis of Feature Models Is Easy. In *SPLC*. Software Engineering Institute, 231–240.
[74] Marcílio Mendonça, Andrzej Wąsowski, Krzysztof Czarnecki, and Donald Cowan. 2008. Efficient Compilation Techniques for Large Scale Feature Models. In *GPCE*. ACM, 13–22.
[75] Matthew W Moskewicz, Conor F Madigan, Ying Zhao, Lintao Zhang, and Sharad Malik. 2001. Chaff: Engineering an Efficient SAT Solver. In *DAC*. ACM, 530–535.
[76] Daniel-Jesus Munoz, Jeho Oh, Mónica Pinto, Lidia Fuentes, and Don Batory. 2019. Uniform Random Sampling Product Configurations of Feature Models That Have Numerical Features. In *SPLC*. ACM, 289–301.
[77] Michael Nieke, Jacopo Mauro, Christoph Seidl, Thomas Thüm, Ingrid Chieh Yu, and Felix Franzke. 2018. Anomaly Analyses for Feature-Model Evolution. In *GPCE*. ACM, 188–201.
[78] Michael Nieke, Gabriela Sampaio, Thomas Thüm, Christoph Seidl, Leopoldo Teixeira, and Ina Schaefer. 2022. Guiding the Evolution of Product-Line Configurations. *SoSyM* 21 (2022), 225–247. Issue 1.
[79] Jeho Oh, Don Batory, Margaret Myers, and Norbert Siegmund. 2017. Finding Near-Optimal Configurations in Product Lines by Random Sampling. In *FSE*. 61–71.
[80] Jeho Oh, Paul Gazzillo, Don Batory, Marijn Heule, and Maggie Myers. 2019. *Uniform Sampling From Kconfig Feature Models*. Technical Report TR-19-02. University of Texas at Austin, Department of Computer Science.
[81] Jeho Oh, Paul Gazzillo, Don Batory, Marijn Heule, and Margaret Myers. 2020. *Scalable Uniform Sampling for Real-World Software Product Lines*. Technical Report TR-20-01. University of Texas at Austin, Department of Computer Science.
[82] Jeho Oh, Necip Fazıl Yıldıran, Julian Braha, and Paul Gazzillo. 2021. Finding Broken Linux Configuration Specifications by Statically Analyzing the Kconfig Language. In *ESEC/FSE*. ACM, 893–905.
[83] Leonardo Passos, Marko Novakovic, Yingfei Xiong, Thorsten Berger, Krzysztof Czarnecki, and Andrzej Wąsowski. 2011. A Study of Non-Boolean Constraints in Variability Models of an Embedded Operating System. In *FOSD (SPLC)*. ACM, Article 2, 8 pages.
[84] Juliana Alves Pereira, Pawel Matuszyk, Sebastian Krieter, Myra Spiliopoulou, and Gunter Saake. 2016. A Feature-Based Personalized Recommender System for Product-Line Configuration. In *GPCE*. ACM, 12 pages.
[85] Tobias Pett, Tobias Heß, Sebastian Krieter, Thomas Thüm, and Ina Schaefer. 2023. Continuous T-Wise Coverage. In *SPLC*. ACM, 87–98.
[86] Tobias Pett, Sebastian Krieter, Tobias Runge, Thomas Thüm, Malte Lochau, and Ina Schaefer. 2021. Stability of Product-Line Sampling in Continuous Integration. In *VaMoS*. ACM, Article 18, 9 pages.
[87] Tobias Pett, Thomas Thüm, Tobias Runge, Sebastian Krieter, Malte Lochau, and Ina Schaefer. 2019. Product Sampling for Product Lines: The Scalability Challenge. In *SPLC*. ACM, 78–83.
[88] Quentin Plazar, Mathieu Acher, Gilles Perrouin, Xavier Devroey, and Maxime Cordy. 2019. Uniform Sampling of SAT Solutions for Configurable Systems: Are We There Yet?. In *ICST*. IEEE, 240–251.
[89] Richard Pohl, Kim Lauenroth, and Klaus Pohl. 2011. A Performance Comparison of Contemporary Algorithmic Approaches for Automated Analysis Operations on Feature Models. In *ASE*. IEEE, 313–322.
[90] Steven David Prestwich. 2009. CNF Encodings. *Handbook of Satisfiability* 185 (2009), 75–97.
[91] David Romero-Organvídez, José A Galindo, Chico Sundermann, Jose-Miguel Horcas, and David Benavides. 2024. Uvlhub: A Feature Model Data Repository Using Uvl and Open Science Principles. *JSS* (2024). To appear.
[92] Pierre-Yves Schobbens, Patrick Heymans, and Jean-Christophe Trigaux. 2006. Feature Diagrams: A Survey and a Formal Semantics. In *RE*. IEEE, 136–145.
[93] Alexander Schultheiß, Paul Maximilian Bittner, Sandra Greiner, and Timo Kehrer. 2023. Benchmark Generation With VEVOS: A Coverage Analysis of Evolution Scenarios in Variant-Rich Systems. In *VaMoS*. ACM, 13–22.
[94] Sandro Schulze, Thomas Thüm, Martin Kuhlemann, and Gunter Saake. 2012. Variant-Preserving Refactoring in Feature-Oriented Software Product Lines. In *VaMoS*. ACM, 73–81.
[95] Sergio Segura, José A. Galindo, David Benavides, José A. Parejo, and Antonio Ruiz-Cortés. 2012. BeTTy: Benchmarking and Testing on the Automated Analysis of Feature Models. In *VaMoS*. ACM, 63–71.
[96] Samuel Sepúlveda and Ania Cravero. 2022. Reasoning Algorithms on Feature Modeling – A Systematic Mapping Study. *Applied Sciences* 12 (2022).
[97] Shubham Sharma, Rahul Gupta, Subhajit Roy, and Kuldeep S Meel. 2018. Knowledge Compilation Meets Uniform Sampling. In *Proc. Int'l Conf. on Logic for Programming, Artificial Intelligence, and Reasoning*. EasyChair, 620–636.
[98] Shubham Sharma, Subhajit Roy, Mate Soos, and Kuldeep S Meel. 2019. GANAK: A Scalable Probabilistic Exact Model Counter. In *IJCAI*, Vol. 19. AAAI Press, 1169–1176.
[99] Steven She, Rafael Lotufo, Thorsten Berger, Andrzej Wąsowski, and Krzysztof Czarnecki. 2011. Reverse Engineering Feature Models. In *ICSE*. ACM, 461–470.
[100] Norbert Siegmund, Marko Rosenmüller, Martin Kuhlemann, Christian Kästner, Sven Apel, and Gunter Saake. 2012. SPL Conqueror: Toward Optimization of Non-functional Properties in Software Product Lines. *SQJ* 20, 3-4 (2012), 487–517.
[101] Stefan Sobernig and Olaf Lessenich. 2021. V1e: A Kernel for Domain-Specific Textual Variability Modelling Languages. In *VaMoS*. ACM, Article 4, 7 pages.
[102] Joshua Sprey, Chico Sundermann, Sebastian Krieter, Michael Nieke, Jacopo Mauro, Thomas Thüm, and Ina Schaefer. 2020. SMT-Based Variability Analyses in FeatureIDE. In *VaMoS*. ACM, Article 6, 9 pages.
[103] Chico Sundermann, Kevin Feichtinger, Dominik Engelhardt, Rick Rabiser, and Thomas Thüm. 2021. Yet Another Textual Variability Language? A Community Effort Towards a Unified Language. In *SPLC*. ACM, 136–147.
[104] Chico Sundermann, Tobias Heß, Dominik Engelhardt, Rahel Arens, Johannes Herschel, Kevin Jedelhauser, Benedikt Jutz, Sebastian Krieter, and Ina Schaefer. 2021. Integration of UVL in FeatureIDE. In *MODEVAR*. ACM, 73–79.
[105] Chico Sundermann, Tobias Heß, Michael Nieke, Paul Maximilian Bittner, Jeffrey M. Young, Thomas Thüm, and Ina Schaefer. 2023. Evaluating State-of-the-Art #SAT Solvers on Industrial Configuration Spaces. *EMSE* 28, 29 (2023), 38.
[106] Chico Sundermann, Elias Kuiter, Tobias Heß, Heiko Raab, Sebastian Krieter, and Thomas Thüm. 2023. On the Benefits of Knowledge Compilation for Feature-Model Analyses. *AMAI* (2023).
[107] Chico Sundermann, Michael Nieke, Paul Maximilian Bittner, Tobias Heß, Thomas Thüm, and Ina Schaefer. 2021. Applications of #SAT Solvers on Feature Models. In *VaMoS*. ACM, Article 12, 10 pages.
[108] Chico Sundermann, Stefan Vill, Thomas Thüm, Kevin Feichtinger, Prankur Agarwal, Rick Rabiser, José A. Galindo, and David Benavides. 2023. UVLParser: Extending UVL With Language Levels and Conversion Strategies. In *SPLC*. ACM, 39–42.
[109] Thomas Thüm. 2020. A BDD for Linux? The Knowledge Compilation Challenge for Variability. In *SPLC*. ACM, Article 16, 6 pages.
[110] Thomas Thüm, Sven Apel, Christian Kästner, Ina Schaefer, and Gunter Saake. 2014. A Classification and Survey of Analysis Strategies for Software Product Lines. *CSUR* 47, 1 (2014), 6:1–6:45.
[111] Thomas Thüm, Don Batory, and Christian Kästner. 2009. Reasoning About Edits to Feature Models. In *ICSE*. IEEE, 254–264.
[112] Marc Thurley. 2006. sharpSAT - Counting Models With Advanced Component Caching and Implicit BCP. In *SAT*. Springer, 424–429.
[113] Leslie G Valiant. 1979. The Complexity of Enumeration and Reliability Problems. *SICOMP* 8, 3 (1979), 410–421.
[114] Ángel Jesús Varela-Vaca, Rafael M. Gasca, Jose Antonio Carmona-Fombella, and María Teresa Gómez-López. 2020. AMADEUS: Towards the AutoMAteD SecUrity TeSting. In *SPLC*. ACM, Article 11, 12 pages.
[115] E.J. Weyuker and F.I. Vokolos. 2000. Experience with Performance Testing of Software Systems: Issues, an Approach, and Case Study. *TSE* 26, 12 (2000), 1147–1156.
[116] John Whaley. 2007. JavaBDD. https://javabdd.sourceforge.net/.
[117] Claes Wohlin, Per Runeson, Martin Höst, Magnus C. Ohlsson, and Björn Regnell. 2012. *Experimentation in Software Engineering*. Springer.
[118] Zhenchang Xing, Yinxing Xue, and Stan Jarzabek. 2013. A Large Scale Linux-Kernel Based Benchmark for Feature Location Research. In *ICSE*. IEEE, 1311–1314.

C.6 TOSEM'24: Reusing d-DNNFs for Efficient Feature-Model Counting

[C2] Chico Sundermann, Heiko Raab, Tobias Heß, Thomas Thüm, and Ina Schaefer. "Reusing d-DNNFs for Efficient Feature-Model Counting". In: *Trans. on Software Engineering and Methodology (TOSEM)* 33.8 (2024). DOI: 10.1145/3680465

Published under Creative Commons Attribution 4.0 International (CC-BY), see https://creativecommons.org/licenses/by/4.0/.



Reusing d-DNNFs for Efficient Feature-Model Counting

CHICO SUNDERMANN, HEIKO RAAB, and TOBIAS HEß, University of Ulm, Ulm, Germany
THOMAS THÜM, Paderborn University, Paderborn, Germany
INA SCHAEFER, Karlsruhe Institute of Technology, Karlsruhe, Germany

Feature models are commonly used to specify valid configurations of a product line. In industry, feature models are often complex due to numerous features and constraints. Thus, a multitude of automated analyses have been proposed. Many of those rely on computing the number of valid configurations, which typically depends on solving a #SAT problem, a computationally expensive operation. Even worse, most counting-based analyses require evaluation for multiple features or partial configurations resulting in numerous #SAT computations on the same feature model. Instead of repetitive computations on highly similar formulas, we aim to improve the performance by reusing knowledge between these computations. In this work, we are the first to propose reusing d-DNNFs for performing repetitive counting queries on features and partial configurations. In our experiments, reusing d-DNNFs saved up-to ~99.98% compared to repetitive invocations of #SAT solvers even when including compilation times. Overall, our tool `ddnnife` combined with the d-DNNF compiler `d4` appears to be the most promising option when dealing with many repetitive feature-model counting queries.

CCS Concepts: • **Computing methodologies** → **Knowledge representation and reasoning**; **Representation of Boolean functions**; • **Software and its engineering** → **Software product lines**; • **Hardware** → *Theorem proving and SAT solving*;

Additional Key Words and Phrases: d-DNNF, feature model, product line, knowledge compilation, model counting, #SAT

ACM Reference format:
Chico Sundermann, Heiko Raab, Tobias Heß, Thomas Thüm, and Ina Schaefer. 2024. Reusing d-DNNFs for Efficient Feature-Model Counting. *ACM Trans. Softw. Eng. Methodol.* 33, 8, Article 208 (November 2024), 32 pages.
https://doi.org/10.1145/3680465

1 Introduction

A product line describes a family of related products which share a set of features [7, 9, 63, 95, 99]. Typically, not every configuration of these features is valid (i.e., represents an actual product). Feature models are the de facto standard for specifying the set of valid software configurations [7, 9, 82] and are also frequently used in other domains [32, 34, 35, 47]. In general, a feature model consists of the underlying features and a set of constraints describing dependencies between features. Due to high numbers of features and constraints, it is typically infeasible to analyze feature models

Authors' Contact Information: Chico Sundermann (corresponding author), University of Ulm, Ulm, Germany; e-mail: chico.sundermann@uni-ulm.de; Heiko Raab, University of Ulm, Ulm, Germany; e-mail: heiko.raab@uni-ulm.de; Tobias Heß, University of Ulm, Ulm, Germany; e-mail: tobias.hess@uni-ulm.de; Thomas Thüm, Paderborn University, Paderborn, Germany; e-mail: thomas.thuem@uni-paderborn.de; Ina Schaefer, Karlsruhe Institute of Technology, Karlsruhe, Germany; e-mail: ina.schaefer@kit.edu.



This work is licensed under a Creative Commons Attribution International 4.0 License.

© 2024 Copyright held by the owner/author(s).
ACM 1557-7392/2024/11-ART208
https://doi.org/10.1145/3680465

manually [9]. Hence, automated analyses, such as checking whether a given configuration is valid, are demanded [2, 9–11, 52, 64, 88].

Numerous practice-relevant analyses depend on computing the number of valid configurations for a feature model [17, 30, 36, 38, 53, 91, 92], such as uniform random sampling [74] and feature prioritization [92]. To employ standardized solvers, feature models are often translated to propositional logic for analysis [21, 66]. While SAT solvers are used for checking the satisfiability, #SAT solvers are dedicated to compute the number of satisfying assignments [53, 93]. Even though #SAT is computationally complex [31, 43, 97, 98], in recent work, we observed [90] that state-of-the-art #SAT solvers can compute the number of valid configurations for many real-world feature models within a few seconds.

For the majority of applications based on feature-model counting, a single #SAT invocation is not sufficient [91]. For example, it can be beneficial to prioritize those features in development that appear in many valid configurations. Using an off-the-shelf #SAT solver as black box, it may be necessary to invoke a separate #SAT computation for every single feature [36, 92]. While several incremental solvers (i.e., that reuse information between queries) for regular SAT were considered in the literature [27, 70], incremental #SAT was not considered yet. As feature models with up to tens of thousands of features have been reported [47, 53, 58, 60, 93], repetitive counting queries without reusing information may require days of computation time [93].

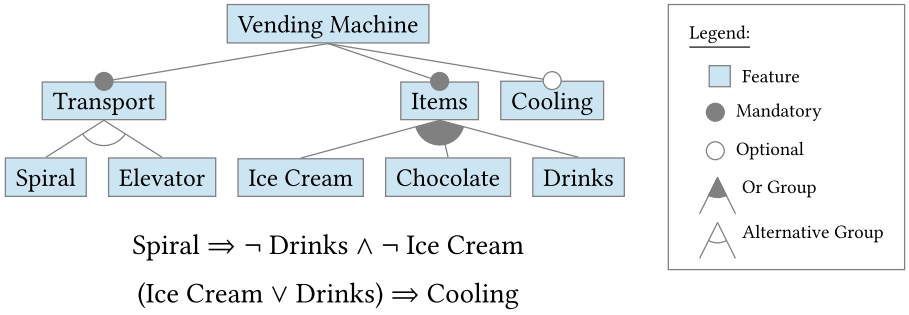
Short response times and reduced resource consumption for feature-model counting are mandated in practice. While long runtimes can be remedied to some degree (e.g., by using parallelization), feature-model analyses are often applied in interactive applications [2, 10, 32, 52, 64] or continuous-integration environments [59, 78]. For instance, many analyses based on feature-model counting can help to understand the configuration space after a change [92] which is relevant for industrial applications [12, 13]. Long response times for analyzing the changes may severely hinder the workflow. We also observed the mandate for short response times from collaborations with the automotive industry, where we already integrated our proposal to improve scalability.

With ever-growing feature models [41, 60, 93], one is therefore interested in reducing the runtime required for analyses. We propose to employ knowledge compilation to reduce the computational effort of multiple queries [24, 26, 56, 68]. Hereby, the original formula is translated to a target format in an offline phase once. If the compilation is successful, the resulting format can be reused for online computations (e.g., SAT or #SAT queries) with typically polynomial-time complexity with respect to the size of the knowledge-compilation artifact [26].

The **deterministic decomposable negation normal form (d-DNNF)** enables computing the number of satisfying assignments in linear time with respect to its number of nodes [22]. Compilation to d-DNNF is similarly complex as solving a #SAT problem as the d-DNNF is constructed from the trace of one #SAT invocation with available tools [24, 56, 68]. In recent work [93], we showed that d-DNNF compilers [24, 40, 56, 68, 76] are generally faster for feature models than other knowledge compilers that enable model counting. For instance, compiling to d-DNNF is substantially faster than compilation to **binary decision diagrams (BDDs)** [39, 90, 91]. Thus, d-DNNFs are a promising target format for feature-model counting.

While the compilation to d-DNNF is well researched [24, 40, 56, 68, 76], reusing a given d-DNNF is only explored sparsely [23, 86]. In particular, reusing a d-DNNF for multiple counting queries has not been evaluated before in software engineering or any other domain. Some publications used d-DNNF compilers for counting as black-box #SAT solvers *without* reusing the d-DNNF, essentially compiling a new d-DNNF for every query [53, 93]. Further, Sharma et al. [86] traverse d-DNNFs to derive satisfying assignments for uniform random sampling.

In this work, we propose to *reuse* d-DNNFs for repetitive counting queries which we apply for computing number of valid configurations that (1) contain a certain feature or (2) conform to a



$$\text{Spiral} \Rightarrow \neg\,\text{Drinks} \wedge \neg\,\text{Ice Cream}$$

$$(\text{Ice Cream} \vee \text{Drinks}) \Rightarrow \text{Cooling}$$

Fig. 1. Feature model example.

partial configuration (i.e., include and/or exclude sets of features). In particular, we compile a given feature model to d-DNNF with a one-time effort using existing compilers [24, 56, 68] and then reuse the d-DNNF for numerous counting-based computations. We showcase the vast improvement of our approach over the state-of-the-art with a large-scale empirical evaluation. Thereby, our work focuses on the efficient reuse of d-DNNFs with the following contributions:

First, we present algorithms and optimizations for repetitive counting queries on a given d-DNNF (cf. Section 4). The basic algorithms are adopted from ideas of Darwiche for counting on d-DNNFs [23]. Still, Darwiche's ideas were not followed up further with an implementation or empirical evaluation yet. Further, we contribute a set of novel optimizations which vastly reduce the number of required traversals through the d-DNNF and also accelerate single traversals.

Second, we provide an open-source tool `ddnnife`[1] including all presented algorithms and optimizations (cf. Section 5). Our d-DNNF reasoner `ddnnife` takes a d-DNNF as input supporting all formats used by the popular compilers [24, 56, 68]. Given the d-DNNF, `ddnnife` allows the user to perform counting queries.

Third, we perform a large-scale empirical evaluation comparing the reuse of d-DNNFs with state-of-the-art #SAT solvers on 61 industrial feature models (cf. Section 6). Hereby, we compare our tool `ddnnife` paired with each available d-DNNF compiler to state-of-the-art solutions. The required data, tools, and framework are publicly available for replicability and reproducibility.[2]

2 Motivating Example

In this section, we show use cases of analyses dependent on multiple counting queries on a small example. Figure 1 shows a feature model representing a simplified vending machine. Each vending machine requires at least one `Item` (denoted as or group) and exactly one `Transport`-system (denoted as alternative group). Furthermore, `Drinks` and `Ice Cream` cannot be sold with a `Spiral`-system, which just drops the items, and `Ice Cream` and `Lemonade` require `Cooling` as denoted by the cross-tree constraints below the tree.

Computing the number of valid configurations enables various analyses. In recent work [91], we presented 18 applications dependent on computing the number of valid configurations (1) collected from the literature [6, 20, 36–38, 53, 92] and (2) inspired from industry collaborations. Those applications are relevant for detecting errors and smells, testing, maintainability, supporting development, economical estimations, and user guidance. 11 of the 18 applications depend on multiple #SAT computations for the same feature model [91].

As an example, the number of valid configurations containing a specific feature can be used to quantify the impact of an error [53, 92]. It may be sensible to address an issue that appears in most

[1] https://github.com/SoftVarE-Group/d-dnnf-reasoner
[2] https://doi.org/10.5281/zenodo.12699708

valid configurations before an issue appearing in only a small fraction of valid configurations. This is particularly relevant if the configurations used in field are unknown as for the Linux kernel [72]. In our example, an error in feature `Elevator` has an impact on eight valid configurations while an error in feature `Spiral` impacts only two valid configurations.

Feature-model counting can also be used to support users during the interactive configuration of a product line [17]. Here, the number of remaining valid configurations can be used as an indicator for the impact of a user selection. Suppose a user already selected `Chocolate` and thinks about whether they want to select or deselect `Cooling`. When selecting `Cooling`, there are still five configurations possible. Otherwise, only two are left which may be an incentive for selecting `Cooling`. Such information may support the entire configurations process requiring model counts for each remaining feature every time the user makes a selection. Here, response time is critical for the usability.

While both presented example analyses require only a few counting queries for our running example, such analyses often require many queries on large feature models in practice. For instance, homogeneity [30, 37, 91, 92], which describes the similarity of configurations induced by the feature model, requires a counting query for every single feature. Especially, when dealing with large industrial feature models containing thousands of features, performing such analyses may induce large runtimes. For each of the 11 applications requiring multiple #SAT invocations, it may be beneficial to reuse d-DNNFs over multiple computations [91, 92].

3 Background

In this section, we provide the background required for the following sections. First, we introduce feature models and feature-model counting. Second, we explain d-DNNFs and how to use their structure for model counting.

3.1 Feature-Model Counting

For a given product line, a feature model is typically used to specify the set of valid configurations [9]. Formally, we define feature models as tuple $FM = (FT, CT)$ with a set of features FT and constraints CT between these features. The set of constraints CT includes both hierarchical constraints that describe the parent-child relationship, and cross-tree constraints. For the sake of simplicity, we consider the constraints to be a conjunction of *propositional* formulas (i.e., each constraint in CT has to be satisfied to satisfy the feature model). We refer to the resulting formula as $F_{FM} = \bigwedge_{c \in CT} c$. It is well-known that each hierarchical constraint (i.e., alternative, or, mandatory, and optional) can be translated to a propositional formula [9, 21].

A configuration $C = (I, E)$ consists of two sets of features which describe features included ($I \subseteq FT$) in the configuration and excluded ($E \subseteq FT$) in the configuration. A feature cannot be included and excluded in the same configuration (i.e., $I \cap E = \emptyset$). If all features are included or excluded (i.e., $I \cup E = FT$), the configuration is complete. Otherwise, the configuration is partial [9]. We consider a configuration $C = (I, E)$ to be valid, iff it is complete and satisfies all constraints CT imposed by the feature model [9]. We refer to the set of all valid configurations induced by FM as VC_{FM}.

Feature-model counting refers to computing the number of valid configurations for a given feature model. In previous work [92], we introduced three types of feature-model counting that compute the number of valid configurations (1) of the entire feature model, (2) that include a certain feature, and (3) that are induced by a partial configuration (i.e., include some and exclude other features). These three types cover all the 21 applications that we identified.

Definition 1 (Cardinality of a Feature Model). The cardinality $\#FM$ of a feature model $FM = (FT, CT)$ corresponds to the cardinality of the set of valid configurations VC_{FM} (i.e., $\#FM = |VC_{FM}|$).

Definition 2 (Cardinality of a Feature). The cardinality $\#f$ of a feature f in a feature model $FM = (FT, CT)$ corresponds to the cardinality of the set of valid configurations that include f (i.e., $\#f = |\{C = (I, E) \mid C \in VC_{FM}, f \in I\}|$).

Definition 3 (Cardinality of a Partial Configuration). The cardinality $\#C_p$ of a partial configuration $C_p = (I_p, E_p)$ given a feature model $FM = (FT, CT)$ corresponds to the cardinality of the set of valid configurations that include each feature $i \in I_p$ and exclude each feature $e \in E_p$ (i.e., $\#C_p = \left|\{C = (I, E) \mid C \in VC_{FM}, I_p \subseteq I, E_p \subseteq E\}\right|$).

Many feature-model analyses are reduced to solving SAT or #SAT problems on a propositional formula representing the feature model [9, 36, 53, 57, 66, 96, 101]. For example, a feature model $FM = (FT, CT)$ is void (i.e., induces no valid configuration [9]) iff the formula F_{FM} is not satisfiable. The three types of cardinality defined above can be reduced to #SAT invocations [36, 92]. The cardinality of a feature model $\#FM$ is equivalent to the number of satisfying assignments of F_{FM} (i.e., $\#\text{SAT}(F_{FM})$). The cardinality of a feature $\#f$ can be computed via $\#\text{SAT}(F_{FM} \wedge f)$. Analogously, the cardinality $\#C_p$ of a partial configuration $C_p = (I, E)$ can be reduced to $\#\text{SAT}(F_{FM} \wedge \bigwedge_{i \in I} i \wedge \bigwedge_{e \in E} \neg e)$.

3.2 Model Counting with d-DNNFs

Typically, tools based on propositional logic, such as SAT and #SAT solvers, rely on normal forms as representation for input formulas. The most commonly used format is **conjunctive normal form (CNF)** [8, 14, 16, 24, 25, 45, 49, 56, 68, 77, 85, 87, 97]. However, CNFs do not support model counting in polynomial time [26, 98].

In contrast, the *d-DNNF* format supports linear-time model counting with respect to the number of d-DNNF nodes [24, 26, 68]. The language d-DNNF is a subset of **negation normal form (NNF)** for which determinism and decomposability holds [22]. Often, d-DNNFs are further restricted to be *smooth* for simplifying and accelerating algorithms on a d-DNNF [23, 25, 26, 68]. All those terms are defined and exemplified below.

Definition 4 (Negation Normal Form [22]). A propositional formula F is in *NNF* iff $\wedge$, $\vee$, $\neg$, $\top$, and $\bot$ are the only logical operators and negations ($\neg$) only appear directly in front of literals, $\top$, and $\bot$.

Algorithm 1 shows how to compute the number of satisfying assignments for a given d-DNNF. Starting from the root, the algorithm traverses the d-DNNF and applies rules depending on the node type (i.e., conjunction, disjunction, and literals) exploiting the properties decomposability, determinism, and smoothness. In the following, we explain these three properties of d-DNNFs and their impact on Algorithm 1.

Definition 5 (Decomposability [24]). A propositional formula F is *decomposable* iff for each conjunction $C = C_1 \wedge C_2 \wedge \ldots \wedge C_n$ the sets of variables of each conjunct C_i are disjunct (i.e., $\forall\, i, j : i \neq j : vars(C_i) \cap vars(C_j) = \emptyset$).

With decomposability, the number of satisfying assignments of a conjunction is the product of its children's numbers of satisfying assignments, as seen in Lines 2–3 of Algorithm 1. If a formula F is decomposable, the conjuncts of each conjunction share no variables. Thus, we can combine the satisfying assignments pairwise (i.e., building the cross-product) for each of the conjuncts. Due to the pairwise combination, the size of the resulting set (i.e., the number of satisfying assignments) is the product of sizes for the respective conjuncts.

Consider the example formula $F = (A \vee B) \wedge (C \vee \neg C)$. The left part $A \vee B$ induces the three satisfying assignments $\{A, B\}$, $\{A, \neg B\}$, $\{\neg A, B\}$. The right part $C \vee \neg C$ induces the two satisfying

Algorithm 1: Model Counting via d-DNNF Traversal

```
1:  procedure #SAT(ddnnf)
2:      return #NODE(ddnnf.root)
3:  procedure #NODE(node)
4:      if node is And then
5:          return ∏_{c∈children} #NODE(c)
6:      else if node is Or then
7:          return ∑_{c∈children} #NODE(c)
8:      else if node is Literal then
9:          return 1
10:     else if node is ⊤ then
11:         return 1
12:     else if node is ⊥ then
13:         return 0
```

assignments $\{C\}, \{\neg C\}$. Pairwise combining both sets results in $3 \cdot 2 = 6$ satisfying assignments for F (i.e., $\{A, B, C\}$, $\{A, B, \neg C\}$, $\{A, \neg B, C\}$, $\{A, \neg B, \neg C\}$, $\{\neg A, B, C\}$, $\{\neg A, B, \neg C\}$). Without decomposability, the described procedure may lead to faulty results as the combined assignments may be conflicting or not disjunct leading to faulty results when counting.

Definition 6 (Determinism [24]). A propositional formula F is *deterministic* iff for each disjunction $D = D_1 \vee D_2 \vee \ldots \vee D_m$, the disjuncts share no common satisfying assignments (i.e., $\forall\, i, j : i \neq j : D_i \wedge D_j = \bot$).

With determinism, the number of satisfying assignments of a disjunction is the sum of its children's numbers as seen in Lines 4–5 of Algorithm 1. For a disjunction, each assignment that satisfies one of the disjuncts also satisfies the disjunction. Therefore, the set of satisfying assignments is the union over the set of the conjuncts. For a deterministic disjunction, there are no solutions shared between disjuncts. Thus, when conjoining the set of satisfying assignments of all disjuncts there are no duplicate entries.

Consider the example formula $F = (A \wedge B) \vee (\neg B)$. $(A \wedge B)$ and $(\neg B)$ induce $A = \{\{A, B\}\}$ and $B = \{\{A, \neg B\}, \{\neg A, \neg B\}\}$ as satisfying assignments, respectively. The set of satisfying assignments for F is the union $A \cup B$ of both sets. As A and B are disjunct by definition, the number of satisfying assignments is the sum of the number of satisfying assignments for its children. Without determinism, the described procedure may lead to faulty results due to duplicate satisfying assignments.

Definition 7 (Smoothness [22]). A propositional formula F is *smooth* iff for each disjunction $D = D_1 \vee D_2 \vee \ldots \vee D_m$, the disjuncts have the same set of variables $(\forall i, j : vars(D_i) = vars(D_j))$.

Smoothness is often enforced for disjunctions to remove the overhead of tracking variables when traversing the d-DNNF [23, 24, 26, 68]. For a smooth disjunction, the sets of variables are equal for every disjunct by definition. Without smoothness, if one of the disjuncts D_i contains a variable v that does not appear in another disjunct D_j, v can be freely assigned (i.e., either $\top$ or $\bot$) in the context of D_j. In this case, the number of satisfying assignments for D_j needs to be multiplied by a factor of two for each missing variable. Consequently, without smoothness, Line 5 in Algorithm 1 would require an adaptation to cope with free variables. As we presume smoothness, the algorithm does not need to keep track of the set of variables. According to Darwiche [22], every d-DNNF can

be smoothed in polynomial time with respect to its number of nodes and variables. In this work, we consider a d-DNNF to be smooth if not stated otherwise.

Due to the presented properties, namely decomposability, determinism, and smoothness, the d-DNNF can be traversed with Algorithm 1 to compute the number of satisfying assignments of a propositional formula [23]. Note that each literal induces exactly one satisfying assignment (i.e., setting the respective variable to $\top$ for f or $\bot$ for $\neg f$) as seen in Lines 6–7 of Algorithm 1. Further, $\top$ induces one satisfying assignment and $\bot$ zero.

The cardinality $\#FM$ of a feature model $FM = (FT, CT)$ is equivalent to the number of overall satisfying assignments $\#F_{FM}$. Hence, the algorithm can be directly used to compute the number of valid configurations for the entire feature model using a d-DNNF representing F_{FM} [86, 90]. In Section 4, we adapt the algorithm to compute the cardinalities of features and partial configurations.

4 Reusing d-DNNFs for Cardinality of Features and Partial Configurations

In this work, we use the benefits of knowledge compilation to d-DNNF for feature-model counting. In particular, we focus on repetitive queries as required for cardinality of features and partial configurations. For a given feature model, we compile it to d-DNNF in a one-time offline phase using existing compilers [24, 56, 68]. Afterwards, we reuse the compiled d-DNNF by reducing the two types of cardinalities queries on this d-DNNF. Those queries can be computed in linear time with respect to the number of nodes in the d-DNNF. The respective d-DNNF queries shown in Section 4.1 are inspired from ideas of an algorithm introduced by Darwiche [23] but have never been implemented nor evaluated. Further, we propose several optimizations to those queries in Section 4.2.

4.1 d-DNNF Queries for Features and Partial Configurations

Algorithms 2 and 3 respectively depict the computation for cardinality of features and partial configurations via d-DNNF traversal. Respectively, the procedures `#FEATURE` and `#CONFIG` compute the cardinality of one feature or configuration by recursively applying rules based on the node type starting from the root node of the d-DNNF. Hereby, both algorithms adapt Algorithm 1 by performing different computations for the literals. Note that Lines 3–11 are exactly equal for both algorithms. Figure 2 shows an example d-DNNF representing the formula $A \wedge ((B \wedge \neg C) \vee (\neg B \wedge C)) \wedge (D \vee \neg D)$[3] as direct acyclic graph.

Each node corresponds to a logical operator or literal. The three numbers next to the nodes represent the number of satisfying assignments of that node for example queries of the three types of cardinalities. In Figure 2, the black numbers refer to traversing the d-DNNF to compute the overall number of satisfying assignments (equivalently: valid configurations) as seen in Algorithm 1. The result in root node of the d-DNNF in Figure 2 indicates a cardinality of $\#F_{FM} = 4$.

Features. To compute the cardinality $\#f$ of a feature $f \in FT$ with $FM = (FT, CT)$, we need to exclude valid configurations $C = (I, E)$ with $C \in VC$ that do not include f (i.e., $f \notin I$). Therefore, when traversing the d-DNNF to compute $\#f$, we discard each satisfying assignment that falsifies f by setting negative literals $\neg f$ to zero. In Figure 2, the blue numbers refer to the cardinality of B. The literal $\neg B$ is initialized with a value of zero. Positive literals of B and other literals are initialized with a value of one. The handling of literals for the cardinality of a feature is described in lines 8–12 of Algorithm 2. Using the adapted literal values, the number of satisfying assignments including B

[3]The tautology $(D \vee \neg D)$ is included to have another simple subformula that is smooth, deterministic, and decomposable. In practice, such structures are often part of d-DNNFs to ensure smoothness (cf. Section 5.1) or to include fully optional variables in the d-DNNF.

Algorithm 2: Cardinality of Features

```
1:  procedure #FEATURE(ddnnf, feature)
2:      return #NODE_FEAT(ddnnf.root, feature)
3:  procedure #NODE_FEAT(node, feature)
4:      if node is And then
5:          return ∏_{c∈children} #NODE(c, feature)
6:      else if node is Or then
7:          return ∑_{c∈children} #NODE(c, feature)
8:      else if node is ⊤ then
9:          return 1
10:     else if node is ⊥ then
11:         return 0
12:     else if node is Literal then
13:         return #LITERAL_FEAT(node, feature)
14: procedure #LITERAL_FEAT(literal, feature)
15:     if literal.variable == feature then
16:         if literal.positive then return 1
17:         else return 0
18:     else return 1
19:
20:
21:
22:
```

Algorithm 3: Cardinality of Partial Configurations

```
1:  procedure #CONFIG(ddnnf, config)
2:      return #NODE_CONFIG(ddnnf.root, config)
3:  procedure #NODE_CONFIG(node, config)
4:      if node is And then
5:          return ∏_{c∈children} #NODE(c, config)
6:      else if node is Or then
7:          return ∑_{c∈children} #NODE(c, config)
8:      else if node is ⊤ then
9:          return 1
10:     else if node is ⊥ then
11:         return 0
12:     else if node is Literal then
13:         return #LITERAL_PC(node, config)
14: procedure #LITERAL_PC(literal, config)
15:     if literal.variable ∈ config.included then
16:         if literal.positive then return 1
17:         else return 0
18:     else if literal.variable ∈ config.excluded then
19:         if literal.positive then return 0
20:         else return 1
21:     else
22:         return 1
```

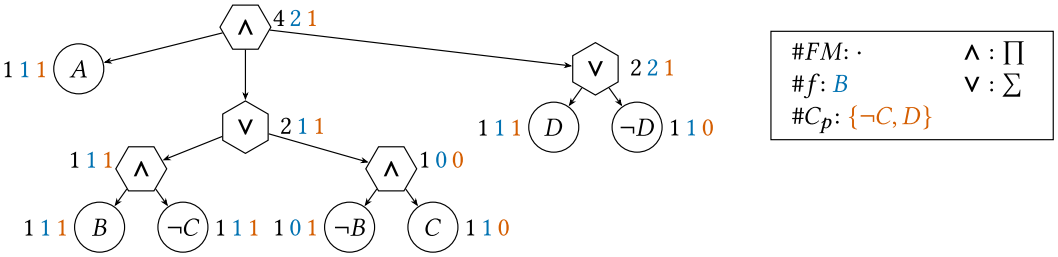


Fig. 2. Computing cardinalities on a d-DNNF.

can be computed by traversing the d-DNNF which results in a cardinality $\#f = 2$ as seen in the root of Figure 2.

Partial Configurations. For the cardinality $\#C_p$ of a partial configuration $C_p = (I_p, E_p)$, we exclude assignments that assign false to an included feature $i \in I_p$ or true to an excluded feature $e \in E_p$. In Figure 2, the orange numbers indicate the cardinality for the partial configuration $\{\neg C, D\}$. For each included feature (here: D), positive and negative literals are initialized with one and zero, respectively. For each excluded feature (here: C), positive and negative literals are initialized with zero and one, respectively. Literals that correspond to features that are neither included nor excluded are always initialized with a value of one. The handling of literals for the cardinality of a partial configuration is specified in Lines 8–16 of Algorithm 3. After initializing the values for literal nodes, the d-DNNF can be traversed to compute the cardinality $\#C_p = 1$ as seen in the root of Figure 2.

Correctness. To illustrate the correctness of our proposed Algorithms 2 and 3, we presume that the original algorithm (Algorithm 1) is correct [23, 26]. The correctness of Algorithm 1 depends on the three properties decomposability, determinism, and smoothness. When applying the assumptions for features and partial configurations, we essentially *condition* the formula for each variable of interest. Alternatively stated, for a decision d we replace each conflicting literal $\neg d$ with $\bot$ (false) and literals d with $\top$ (true). Hence, the adapted formula represents the subset of satisfying assignments that adheres to the assumptions. For the correctness of Algorithms 2 and 3, it remains to show that the adapted formula is still (1) decomposable, (2) deterministic, and (3) smooth after applying the assumptions. First, decomposability is never violated as we only remove variables. It follows that the sets of variables for each conjunct of an $\wedge$-node are still disjunct afterwards. Second, each deterministic $\vee$-node is still deterministic. With decision propagation, we never add new satisfying assignments but only remove them. Hence, two sets of satisfying assignments that were disjunct prior to applying the assumptions are still disjunct. Third, for a smooth $\vee$-node, we remove the same set of variables for every child node. Hence, if the set of variables were equal before applying the assumptions, the set of variables stay equal after. Overall, Algorithms 2 and 3 yield correct results since the adapted formula (1) represents the set of satisfying assignments conforming to the query and (2) is still a smooth d-DNNF.

Summary. With the presented algorithms, d-DNNFs can be reused to compute results for every application based on the cardinality of features and partial configurations [91]. Each described algorithm is an online computation on a given d-DNNF and has a linear runtime with respect to the number of d-DNNF nodes.

4.2 Optimizations

For the algorithms presented in Section 4, every node of a given d-DNNF needs to be traversed for each query. Even though the complexity per query is linear in the number of nodes, analyses in practice may induce a high number of queries on d-DNNFs which may be exponentially larger than the CNF and contain millions of nodes in practice. Further, the algorithm requires processing large numbers with up to thousands of digits in each step [93]. We propose several optimizations to accelerate d-DNNF traversals to answer queries and reduce the number of required traversals.

Partial Traversals. With partial traversals, we aim to accelerate the d-DNNF traversals by reusing results of the overall model count ($\#FM$ in Figure 2). If we compute the number of satisfying assignments under a partial assignment of literals (e.g., for the cardinality of a feature), we only have to recompute the value of nodes that are influenced by a corresponding literal. Hence, for each query, we first traverse the nodes of a d-DNNF to identify nodes that are part of a direct path from a corresponding literal to the root and mark these nodes. Afterwards, we traverse the d-DNNF and only recompute the count of marked nodes. To reuse values that do not change due to a given partial assignment, we cache the original value (i.e., without partial assignment) for every node. Figure 3 shows the partial traversal (blue path) of computing the cardinality of B for our running example. The traversal only requires adapting the values from a direct path between $\neg B$ and the root $\wedge$. Note that for every unaffected/uncolored node, the value for the cardinality of B and the original value is equal. This also applies to the node B which still induces exactly one satisfying assignment when conditioned with the query B. Hence, only four out of the twelve overall nodes need to re-evaluated with our partial traversal.

Iterative Traversal. In practice, d-DNNF nodes often have several parents [56]. Hence, a recursive descent could either redundantly traverse some subtrees multiple times or require overhead to

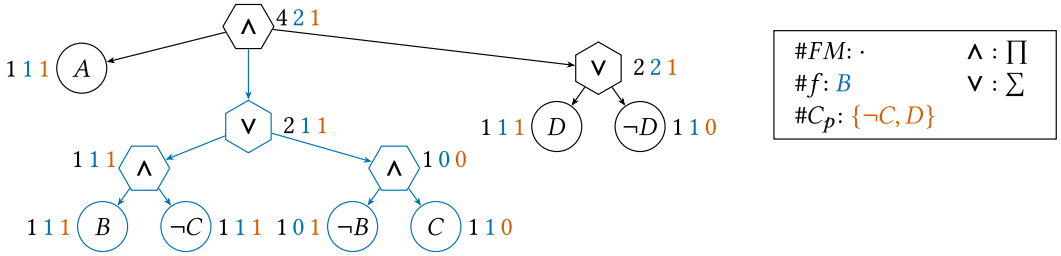


Fig. 3. Partial traversal on a d-DNNF for the cardinality of feature B.

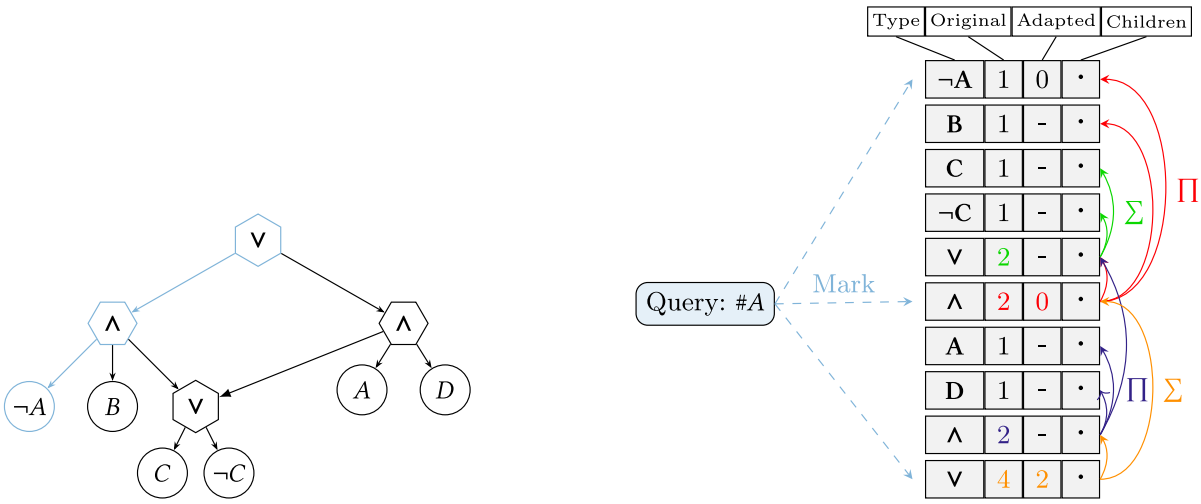


Fig. 4. List format for iterative traversal with sharing.

prevent this. To simplify reusing values and overall reduce the computational effort when traversing the d-DNNF, we use an iterative approach instead of a recursive one. The nodes are stored in a list such that a child is always traversed before its parents. Thus, when processing a node, the values of its children are already available and do not require any recursive descent. Figure 4 shows the list representation we use for the d-DNNF seen on the left side. From left to right, the list representation shows the type of the node (i.e., $\wedge$), the original count, the adapted count, and pointers to its children. To evaluate a query (here: $\#A$) on the d-DNNF, the list is traversed iteratively starting from the first entry in the list. In this case, we start with the node representing the literal $\neg A$. When reaching one of the nodes depending on the values of other nodes, all required values are already computed. For instance, when reaching the first OR node, we just look up the values for C and $\neg C$.

For partial traversals, we ensure to visit only marked nodes. The dashed blue lines indicate which nodes were marked during the first phase for the query $\#A$. Afterwards, the algorithm recomputes the value for each of the three marked nodes. The result for the query $\#A = 2$ is then saved in the adapted value of the root node.

Partial Calculations. When computing the value of a node during a partial traversal, it is possible only for a small subset of its child nodes to be changed. In this case, not recomputing the entire node but adapting the cached value may accelerate the d-DNNF traversal. For instance, given a $\wedge$-node with ten children for which only one child c' changed, we do not build the product of the ten children. Instead, we adapt the original value based on the changed values. If more than half of the children changed, we recompute the value as shown in Algorithm 1. Otherwise, we divide the cached value by the old value of c_{old} and then multiply it with the new value for every changed

child. Analogously, we subtract the cached value by the old value of c_{old} and then add the new value for $\vee$ nodes. The partial calculation requires two arithmetic operations per changed node. For instance, recomputing an $\wedge$-node requires one multiplication and one division. Hence, we only apply partial calculation if less than half of the children values changed.

Core and Dead Features. The knowledge about core and dead features can be used to reduce the number of required queries. Core and dead features appear in all and no valid configurations, respectively [9]. Hence, the cardinality of core and dead features can be directly derived without traversing the d-DNNF again as seen in Equation (1). For a core feature, the number of satisfying assignments is always equal to the number of overall satisfying assignments. The cardinality of a dead feature is always zero. Further, we can also use core and dead features to compute the cardinality of a partial configuration without traversal in some cases. Each time a dead feature is included or a core feature is excluded in a partial configuration, this configuration is unsatisfiable and does not need to be evaluated with a d-DNNF traversal. Further, included core features and excluded dead features can be disregarded for the marking phase of partial traversals.

We propose to compute core and dead features in constant time per feature by exploiting the smoothness of a d-DNNF. With our proposal, we can identify all core and dead features with *one* traversal through the literal nodes using the following properties. A feature f is core iff only positive literals f appear in the d-DNNF. Analogously, a feature f is dead iff only negative literals $\neg f$ appear. In our running example Figure 2, the feature A is core as only positive literals appear in the smooth d-DNNF

$$\#f = \begin{cases} \#FM & \text{if } f \text{ is core} \\ 0 & \text{if } f \text{ is dead} \\ \#f & else \end{cases}. \tag{1}$$

In the following, we provide a short intuition on the correctness of our approach that detects core and dead features. Assuming that our algorithms for the cardinality of partial configurations are correct, the correctness of our proposal using core/dead features is directly implied as explained in the following. We start the explanations with core features. If only positive literals of a variable v appear, applying Algorithm 2 for $\#v$ (i.e., cardinality of v) always produces the same results as the d-DNNF under no assumptions. The only case resulting in a change would be a negative literal of v which conflicts our assumption of only having positive literals of v. Hence, the number of satisfying assignments v appears in is equal to the number of overall satisfying assignments making v a core feature. This works analogously for dead features. If only negative literals of a variable w appear, evaluating $\#(\neg w)$ with Algorithm 2 always yields the same result as for the d-DNNF without assumptions. Thus, there are no satisfying assignments including w in this case as otherwise the results would differ.

Partial Derivatives. For the cardinality of features, we can apply partial derivatives on d-DNNFs as presented by Darwiche [23] instead of querying the d-DNNF for each feature. A partial derivative indicates the impact on the overall result of changing one of the variables under the assumption that every other variable is fixed [23]. As cardinality of features relies on flipping single literals for each computation, we can employ partial derivatives as optimization. The idea is to consider the d-DNNF as computation graph with each $\wedge$ being a multiplication and each $\vee$ being an addition. Here, we can annotate the different nodes with partial derivatives that then enable to compute the cardinality of a feature in constant time. Partial derivatives are not generally applicable for partial configurations, since flipping multiple literals is typically required.

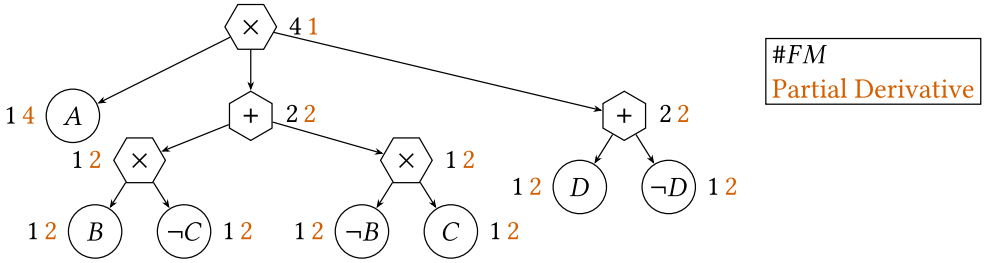


Fig. 5. Computation graph with partial derivatives for d-DNNF shown in Figure 2.

The partial derivatives can be computed by traversing the d-DNNF twice. In the first traversal, each node is annotated with its model count as done in Algorithm 1. In the second traversal, these counts are used to derive the partial derivatives $\frac{\delta F}{\delta n}$ of node n in the context of formula F as shown in Equation (2). Here, $P(n)$ is the set of parents of n, $C(p)$ the set of children of p, and $\#c$ the model count of the node c

$$\frac{\delta F}{\delta n} = \begin{cases} 1 & \text{n is root} \\ \sum_{p \in P(n)} PD(n,p) & \text{else} \end{cases} \tag{2}$$

$$PD(n,p) = \begin{cases} \frac{\delta F}{\delta p} & \text{p is } \vee \\ \frac{\delta F}{\delta p} \cdot \prod_{c \in C(p) \setminus \{n\}} \#c & \text{p is } \wedge \end{cases}. \tag{3}$$

Figure 5 shows the computation graph for Figure 2 annotated with the partial derivatives. Given the partial derivatives, the cardinality of each feature can be computed with Equation (4). Here, $\#FM$ is the original count, $\Delta(f)$ is the change of the variable (e.g., -1 for setting f from true to false), and $\frac{\delta F}{\delta f}$ is the partial derivative of f. For instance, setting A to false reduces the overall model count by 4 as indicated by the orange number in Figure 5

$$\#f = \#FM + \Delta(f) \cdot \frac{\delta F}{\delta f}. \tag{4}$$

Summary. Each of the five presented optimizations reduces either the number of required queries or reduces effort for a single traversal. Both of those impacts potentially improve the efficiency of reusing d-DNNFs for computing the cardinality of features or partial configurations. In Section 6, we examine the general performance of our approach and also take a closer look on the effect of specific optimizations.

5 Tool Support: `ddnnife`

The presented algorithms and optimizations are implemented in our open-source d-DNNF reasoner `ddnnife`, the Swiss *knife* for reusing d-DNNFs. `ddnnife` is implemented in Rust[4] as Rust generally compiles to efficient target code and has great support for improving memory safety. The source code and example input data are available at GitHub[5] under LGPLv3[6] license which allows usage of `ddnnife` with only few restrictions even in commercial settings. We use `ddnnife`, amongst other tools, to evaluate the advantages of reusing d-DNNFs in Section 6.

[4] https://www.rust-lang.org/
[5] https://github.com/SoftVarE-Group/d-dnnf-reasoner
[6] https://www.gnu.org/licenses/lgpl-3.0.de.html

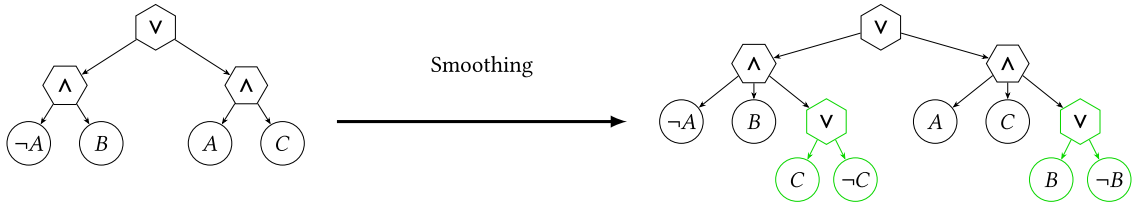


Fig. 6. Smoothing a d-DNNF.

5.1 Parsing d-DNNFs

Input Format. `ddnnife` supports the standard format set by `c2d` [24] and the format introduced by `d4` [56]. Supporting both formats, `ddnnife` is able to parse d-DNNFs from each popular d-DNNF compiler, namely `c2d` [24, 68], `dSharp` (also uses `c2d` format), and `d4` [56]. In addition, `ddnnife` supports CNFs in DIMACS-format proposed by the **Discrete Mathematics and Theoretical Computer Science (DIMACS)** group[84] as input. Here, `d4` is internally used to translate the CNF to d-DNNF.

Listing 1 shows our running example Figure 2 in the `c2d` format.[7] The first line `nnf v e n` indicates the number of nodes (`v`), the number of edges (`e`), and the number of variables (`n`). Each remaining line describes one node in the d-DNNF. `L 1` introduces a positive literal of the variable `1`. An and node is described by the line `A n i j k` with `n` children and `i`, `j`, `k` being indices of the lines holding child nodes. Note that the first line containing a node has the index zero. Or nodes come with an additional parameter that indicates a decision variable (i.e., a variable that is always true in the left and false in the right child). `O d n i j k` describes an or node with the decision variable `d`, `n` child nodes with the indices `i`, `j`, and `k`. If `d` is `0`, there is no decision variable.

Listing 2 shows our running example Figure 2 in the `d4` format.[8] Here, each line either describes a node or an edge between nodes and is terminated with a `0`. Lines describing nodes always first specify the type of node, namely or (o), and (a), true (t), false (f), and then the index of this node. In contrast to the `c2d`-format, the indices do not refer to the line index but to the index specified for each node. Edges always follow the format `p c x y z 0` with the parent node `p`, the child node `c`, and an arbitrary number of literals `x y z` set to true when following this path. For instance, Line 4 specifies that the node with the index `1` has a child node Φ_3 with the index `3`. Further, the literals `2` and `−3` are conjoined resulting in the subformula $(\Phi_3 \wedge (2 \wedge -3)) \vee \Phi_{rest}$ with Φ_{rest} being the formula resulting from other edges starting from node `1`.

Smoothing. As discussed in Section 3.2, computing the cardinalities on an unsmooth d-DNNF causes computational overhead as the algorithm needs to keep track on the set of variables during the traversal [26]. While `c2d` and `dSharp` provide a functionality to produce smooth d-DNNFs, `d4` does not guarantee smoothness. Consequently, the procedure described below is mainly valuable for d-DNNFs compiled by `d4`. Note that similar procedures are used in `dSharp` and `c2d` but are only sparsely documented [24, 68].

To ensure smoothness, we add designated nodes for each unsmooth $\vee$-node if needed, as seen in Figure 6 in a preprocessing step. For each missing variable $v \in missingVariables_\Phi$ in a child subtree Φ of the $\vee$-node, we add a disjunction $v \vee \neg v$ resulting in the following smooth disjunction on the right side

$$\bigvee_{\Phi \in children} \Phi \quad \rightarrow \quad \bigvee_{\Phi \in children} \left(\Phi \wedge \bigwedge_{v \in missingVariables_\Phi} v \vee \neg v \right).$$

[7] http://reasoning.cs.ucla.edu/c2d/
[8] https://github.com/crillab/d4

Listing 1. c2d Format

```
 1  nnf 11 11 4
 2  L 1 # A
 3  L 2 # B
 4  L -3 # !C
 5  L -2 # !B
 6  L 3 # C
 7  L 4 # D
 8  L -4 # !D
 9  A 2 1 2
10  A 2 3 4
11  O 0 2 7 8
12  O 4 2 5 6
13  A 3 0 9 10
```

Listing 2. d4 Format

```
 1  o 1 0
 2  o 2 0
 3  t 3 0
 4  a 4 0
 5  1 3 2 -3 0
 6  1 3 -2 3 0
 7  2 3 4 0
 8  2 3 -4 0
 9  4 3 1 0
10  4 0 2 0
11  4 0 1 0
```

The added subformulas are tautologies as $v \vee \neg v \equiv \top$ holds for every variable v. Hence, when adding smoothing nodes to a subformula Φ, the result $\Phi \wedge \top$ equivalent to Φ. In the example in Figure 6, the left subtree does not include the variable C. Analogously, the right subtree does not include B. Hence, the set of variables of the subtrees are not equal, which violates smoothness. We conjunct a smoothing node $C \vee \neg C$ and $B \vee \neg B$ to the subtrees, respectively. After the smoothing step, both sides of the root $\vee$-node include the same set of variables $\{A, B, C\}$ without changing the semantics of the formula.

Pre-Processing. Prior to performing queries for features or partial configurations, `ddnnife` performs some pre-processing steps to simplify the computation for those queries. First, the given d-DNNF is smoothed as described above. Second, connections from child to parent node are added to simplify the marking algorithm for partial traversals. Third, pointers to the indices of literal nodes are saved to be used as starting point for marking. Fourth, core and dead features are identified by examining if a variable only appears as positive (core) or negative (dead) literal, respectively. Fifth, the model count for each d-DNNF node is computed under no assumptions. Those values are then re-used for partial traversals. After the different the pre-processing steps, the resulting representation of the d-DNNF is then stored and used for querying.

5.2 Implementation of Algorithms

`ddnnife` supports the algorithms for computing the cardinality of feature models, features, and partial configurations presented in Section 4.1. By default, we apply partial *derivatives* for cardinality of features and partial *traversals* for partial configurations. As the parsing step ensures that each d-DNNF is smooth, the algorithms do not keep track of variables during the computation.

Optimizations. `ddnnife` includes all optimizations presented in Section 4.2. However, some optimizations are slightly adapted according to observations when applying `ddnnife` to d-DNNFs representing feature models.

Partial calculation is only applied to $\wedge$-nodes, due to the following two reasons. First, multiplications are more computationally expensive than addition resulting in smaller runtime profits when reducing the number of additions. Second, in practice, $\vee$-nodes typically have exactly two child nodes due to the construction of d-DNNFs with `c2d`, `dSharp`, and `d4`. For exactly two children, the partial-calculation optimization introduces a small overhead but provides no benefit as still two arithmetic operations are required when only one child changes.

When dealing with large partial configurations (i.e., many included or excluded features), a large portion of the d-DNNF changes and needs to be traversed. Thus, if the size of the partial configurations exceeds a certain threshold, the overhead of marking nodes to recompute is no longer worth it. In this case, we do not use partial traversals but re-evaluate the whole d-DNNF.

5.3 Tool Usage

Prior to employing different analyses, `ddnnife` always demands a d-DNNF, either in `c2d` (also used by `dSharp`) or `d4` format, as input. In this section, we shortly explain the different modes of usage for `ddnnife`, namely a one-time computation for a given set of queries and streaming mode for interactive usage. For more details on syntax and technical features, we refer to the repository.[9]

One-Time Computation. When invoking d-DNNF as one-time computation, the user has four options to denote queries: (1) compute cardinality of a feature, (2) cardinality of all features, (3) cardinality of a partial configuration, or (4) a set of different queries given in a text file. After invocation, the tool provides the results either as `.csv` or command line output and then terminates.

Interactive Mode. The interactive mode of `ddnnife` can be used for interactive querying if not all queries are known in advance. At the beginning, `ddnnife` loads and pre-processes the d-DNNF in a one-time effort and waits for queries on features or partial configurations. The interactive mode follows a strict protocol to allow users integrating `ddnnife` in non-rust based tools. Nevertheless, `ddnnife` can also be used interactively with human input. After loading, the user can query the d-DNNF (e.g., `count v 1` to count the cardinality of variable 1). For details on the protocol, we also refer to the repository.

Options. Besides usage mode and parameters for specifying queries, `ddnnife` provides some additional parameters to control its behavior. Often, pre-processing and in particular smoothing the d-DNNF requires a considerable initial effort. Hence, we provide an option to save the smoothed d-DNNF for later usage. Multithreading can also be enabled by the user to answer multiple queries in parallel. In some cases, `d4` omits fully optional features in the d-DNNF. When disregarding those variables, two problems emerge. First, the number of satisfying assignments would be off by two to the power of omitted variables as those can be assigned arbitrarily. Second, `ddnnife` could not compute the cardinality of those features or partial configuration including/excluding one of those as they are not part of the d-DNNF. Hence, we provide a parameter for declaring omitted variables to prevent faulty results.

Summary. For a given d-DNNF, `ddnnife` can be used interactively or with a one-time computation to perform repetitive counting queries. With popular compilers, such as `d4` [56], `c2d` [24], and `dSharp` [68], we can translate a feature model to d-DNNF to perform feature-model counting with `ddnnife`. In Section 6, we examine the performance of `ddnnife` and its optimizations when applied to feature models. Hereby, we focus on the mode for one-time computations.

6 Evaluation

In our empirical evaluation, we analyze the benefits of reusing d-DNNFs for computing the cardinality of features and partial configurations compared to the state of the art. The applicability of applying d-DNNFs for feature-model counting depends on the scalability of both the compilation (offline phase) and the reuse of the compiled d-DNNF with a reasoner (online phase). Hence, we consider both phases in our empirical evaluation. We provide a publicly available replication

[9] https://github.com/SoftVarE-Group/d-dnnf-reasoner

package[10] that includes all evaluated tools, subject systems (except nine confidential systems of our industry partner), and the required tooling for measurements.

6.1 Research Questions

With our research questions, we aim to inspect the advantages of reusing d-DNNFs compared to repetitive #SAT calls. First, we check if using d-DNNFs is generally *applicable* by studying the scalability of available d-DNNF compilers (RQ1). Second, we empirically examine the *correctness* of considered tools (RQ2). Third, we evaluate the runtime *improvements* when reusing d-DNNFs compared to the state-of-the-art (RQ3). Fourth, we compare the available options for reusing d-DNNFs to find *promising candidates* for usage in practice (RQ4).

RQ1: How efficient are d-DNNF compilers when applied to industrial feature models regarding runtime and memory?

To apply our algorithms on industrial feature models, it is necessary to be able to compile the respective formulas to d-DNNF first. With RQ1, we inspect and compare the scalability of d-DNNF compilers regarding runtime and size of the resulting d-DNNFs. We also consider the sizes, as the runtime of `ddnnife` depends on the size of the d-DNNF. While the general performance of the compilers have been considered in other work, the existing evaluations either (1) do not evaluate on feature models [15, 24, 56, 68], (2) use d-DNNF compilers as black-box #SAT solvers [53, 90], or (3) do not consider the size of the d-DNNFs [91]. In particular, the sizes of d-DNNFs representing feature models have not been studied.

RQ2: Which tools and tool combinations compute correct cardinalities?

Correctness is another vital aspect for applicability and also for sound performance comparisons. To examine the correctness of considered tools and in particular our approach `ddnnife`, we compare computed cardinalities.

RQ3: How efficient is reusing d-DNNFs for computing cardinalities of features and partial configurations compared to repetitive #SAT calls?

After examining the general applicability with RQ1 and RQ2, we inspect the main focus of this work (i.e., advantages of reusing d-DNNFs for feature-model counting) with RQ3. We compare runtimes of d-DNNF-based approaches with the runtimes of repetitive #SAT calls for computing cardinality of features and partial configurations. Hereby, we consider reusing d-DNNFs with and without compilation time for comparison.

RQ4: Which d-DNNF-based tools perform best for computing cardinalities?

After evaluating the difference between d-DNNF-based approaches and the state-of-the-art in RQ3, we focus on comparing the different d-DNNF-based tools in RQ4. In particular, we aim to provide recommendation on which combinations of compiler and reasoner to use.

6.2 Experiment Design

Subject Systems. In our empirical evaluation, we only consider industrial feature models and no artificial ones. Artificially generated feature models may result in runtimes that are not representative for the real world as observed previously [5, 39]. With our selection of subject systems, we aim for a wide coverage of industrial feature models from a variety of domains. To this end, we mostly use publicly available feature models used in benchmarks in related work. Table 1 gives an

[10] https://zenodo.org/doi/10.5281/zenodo.12699707

Table 1. Overview Subject Systems (Sorted by Domain and Variables)

System [Source]	Domain	Variables	Clauses	Cardinality
lrzip [74]	Archiver	20	63	144
7z [74]	Archiver	44	210	68,640
Automotive03a [confidential]	Automotive	384	1,020	5.16×10^{10}
Automotive06a [confidential]	Automotive	449	1,339	1.13×10^{10}
Automotive04 [confidential]	Automotive	531	5,961	2.45×10^{21}
Automotive07a [confidential]	Automotive	581	2,235	4.34×10^{15}
Automotive03 [confidential]	Automotive	588	4,638	2.54×10^{31}
Automotive05 [confidential]	Automotive	1,663	45,795	unknown
Automotive01 [50]	Automotive	2,513	10,275	5.43 + 217
Automotive03b [confidential]	Automotive	4,902	9,922	13,984
Automotive06b [confidential]	Automotive	5,571	11,261	17,280
Automotive07b [confidential]	Automotive	8,649	23,631	1.52×10^{8}
Automotive2.4 [47]	Automotive	18,616	350,221	1.65×10^{1534}
JavaGC [74]	Compiler	39	105	193,536
BerkeleyDBC [74]	Database	18	29	2,560
featureide-berkeleydb [64]	Database	76	141	4.08×10^{9}
FinancialServices 2018-05-09 [71]	Finance	771	7,241	9.74×10^{13}
HiPAcc [74]	Systems Software	31	104	13,485
axtls.2.1.4 [74]	Systems Software	94	190	1.96×10^{12}
axTLS [47]	Systems Software	96	203	8.26×10^{11}
fiasco.17.10 [74]	Systems Software	234	1,178	1.02×10^{10}
uClibc-ng.1.0.29 [74]	Systems Software	269	1403	8.02×10^{36}
uClibc [47]	Systems Software	313	1,285	1.66×10^{40}
toybox.0.7.5 [74]	Systems Software	316	108	1.43×10^{81}
uClinux-base [47]	Systems Software	380	7,366	2.61×10^{022}
toybox [74]	Systems Software	544	1,020	1.44×10^{017}
axTLS [74]	Systems Software	684	2,155	4.28×10^{020}
busybox-1.18.0 [47]	Systems Software	854	1,322	2.06×10^{201}
busybox.1.28.0 [79]	Systems Software	998	962	1.31×10^{248}
am31.sim [47]	Systems Software	1,178	2,845	2.62×10^{118}
embtoolkit [47]	Systems Software	1,179	5,967	5.13×10^{096}
ref4955 [74]	Systems Software	1,218	3,099	8.20×10^{123}
ecos-icse11 [74]	Systems Software	1,244	3,146	4.97×10^{125}
pati [74]	Systems Software	1,248	3,266	7.90×10^{126}
p2106 [47]	Systems Software	1,262	3,102	3.02×10^{124}
integrator.arm9 [74]	Systems Software	1,267	50,606	4.05×10^{129}
olpce2294 [74]	Systems Software	1,274	3,881	8.18×10^{126}
adderII [74]	Systems Software	1,276	3,206	3.57×10^{125}
at91sam7sek [74]	Systems Software	1,296	3,921	2.58×10^{127}
se77x9 [74]	Systems Software	1,319	49,937	1.20×10^{135}
m5272c3 [74]	Systems Software	1,323	3,297	2.36×10^{125}
phycore229x [74]	Systems Software	1,360	4,026	1.76×10^{136}
freebsd-icse11 [74]	Systems Software	1,396	62,183	8.38×10^{313}
ea2468 [47]	Systems Software	1,408	3,470	4.81×10^{130}
uClinux-distribution [47]	Systems Software	1,580	2,131	4.07×10^{409}
fiasco [74]	Systems Software	1,638	5,228	3.58×10^{14}
uClinux [74]	Systems Software	1,850	2,468	1.62×10^{091}
linux-2.6.33.3 [47]	Systems Software	6,467	132,032	unknown
busybox-1.18.0 [74]	Systems Software	6,796	17,836	8.49×10^{216}
2.6.28.6-icse11 [74]	Systems Software	6,888	343,944	unknown
uClinux-config [74]	Systems Software	11,254	31,637	7.78×10^{417}
buildroot [74]	Systems Software	14,910	45,603	unknown
embtoolkit [74]	Systems Software	23,516	180,511	2.10×10^{218}
freetz [74]	Systems Software	31,012	102,705	unknown
2.6.32-2var [74]	Systems Software	60,072	268,223	unknown
2.6.33.3-2var [74]	Systems Software	62,482	273,799	unknown
Dune [74]	Solving	17	16	2,304
Polly [74]	Team Management	40	100	40,000
X264 [74]	Video	16	11	1,152
VP9 [74]	Video	42	104	216,000
JHipster [74]	Web Application	45	104	26,256

overview on the subject systems considering domain, number of features, number of clauses, and cardinality.

First, we include feature models from a benchmark[11] provided by Knüppel et al. [47] which consists of various CDL[12] models, KConfig[13] models, and an automotive product line. For the 117 CDL feature models, we observed in previous work [90] that they are highly similar considering various metrics, including number of features, number of constraints, and also performance of #SAT solvers. To prevent a bias, we decided to include only three CDL feature models instead of all 117 in our benchmark. Here, we took the feature models with the minimum, median, and maximum number of features. Second, our benchmark includes feature models from the systems software and application domain[14] provided by Oh et al. [75]. Third, we include a version of the BusyBox feature model[15] provided by Pett et al. [79]. Fourth, we consider a feature model[16] extracted from the financial services domain [71, 80]. Fifth, we consider another feature model from the automotive domain published by Kowal et al. [50]. Sixth, we include a database feature model publicly available in the FeatureIDE examples.[17] Sixth, we include nine feature models representing automotive product lines from an industry collaboration, which unfortunately cannot be published. Overall, our benchmark consists of 62 industrial feature models.

Evaluated Tools. For the selection of tools, our goal is to compare d-DNNF compilation and reuse to the state-of-the-art of feature-model counting. Currently, the most scalable solution that is generally applicable appears to be translating a feature model to CNF and invoking a #SAT solver as black box [69, 74, 93]. As representative of the state-of-the-art, we include the fastest publicly available off-the-shelf #SAT solvers [16, 87, 93, 97]. For representative results, we aim to include each publicly available d-DNNF compiler [24, 56, 68] (i.e., tools compiling CNF to d-DNNF) and reasoner (i.e., tools using d-DNNF to compute the number of satisfying assignments).

Table 2 provides an overview on the tools we identified. Every listed tool is included in our replication package. In a report of the model counting competition 2020 (MC2020), Fichte et al. [31] list existing tools capable of computing the number of satisfying assignments including d-DNNF compilers. In our empirical evaluation, we consider each d-DNNF compiler appearing in their list, namely `c2d` [24], `dSharp` [68], and `d4` [56]. For d-DNNF reasoners, no tool capable of reusing d-DNNFs for repetitive queries have been empirically evaluated or considered in another scientific publication, to the best of our knowledge. However, we extended our search to tools without scientific publication and found two tools providing such functionality, namely the reasoner `query-ddnnf` developed by Caridriot et al.[18] and a new version of `d4` on GitHub. The extended (compared to the original work [56]) version of `d4`, does not allow reasoning on an existing d-DNNF, but allows to compute the number of satisfying assignments for several queries for a given CNF which we also include in our evaluation (referred to as `d4-query`). We evaluate our proposal `ddnnife`, `query-ddnnf` and `d4-query` as d-DNNF-based reasoners. In preliminary experiments, we found that using `query-ddnnf` in combination with `dSharp` always results in cardinalities of zero which may be caused by `dSharp` violating the standard specified by `c2d` in some cases. Thus, we exclude the combination of `query-ddnnf` and `dSharp`. `query-ddnnf` does not support the output format of

[11] https://github.com/AlexanderKnueppel/is-there-a-mismatch
[12] https://ecos.sourceware.org/
[13] https://www.kernel.org/doc/html/latest/kbuild/kconfig-language.html
[14] https://github.com/jeho-oh/Smarch
[15] https://github.com/PettTo/Measuring-Stability-of-Configuration-Sampling
[16] https://github.com/PettTo/SPLC2019_The-Scalability-Challenge_Product-Lines
[17] https://github.com/FeatureIDE/FeatureIDE/tree/v3.11.1/plugins/de.ovgu.featureide.examples/featureide_examples
[18] https://www.cril.univ-artois.fr/KC/d-DNNF-reasoner.html

Table 2. Overview Evaluated Tools

Solver	Type	Reference	Version/Commit
countAntom	#SAT Solver	[16]	1.0
Ganak	#SAT Solver	[87]	18af636[20]
sharpSAT	#SAT Solver	[97]	1.1
c2d	d-DNNF Compiler	[24, 25]	2.2
d4	d-DNNF Compiler	[56]	v2 (v1 for query-ddnnf)
dSharp	d-DNNF Compiler	[68]	5a4490[21]
ddnnife	d-DNNF Reasoner	This Submission	abfaa7[22]
query-ddnnf	d-DNNF Reasoner	No scientific publication	0.4
d4-query	d-DNNF Reasoner	No scientific publication	v2

the latest d4 version. Therefore, we evaluate query-ddnnf with an earlier version (d4v1).[19] If not stated otherwise, d4 refers to the latest version (d4v2) during the evaluation.

As baseline for evaluating our approach using d-DNNFs, we include the #SAT solvers sharpSAT, countAntom, and Ganak. In recent work [90], we identified those three solvers as the fastest #SAT for analyzing feature models. Also, Ganak won the model counting competition 2020 [31], while sharpSAT won in 2021 and 2022.[23] Only countAntom and our tool ddnnife support multi-threading. Since other tools could also benefit from parallelization for the queries, we decided to evaluate every tool with a single thread for better comparability.

The d-DNNF compilers and #SAT solvers require a CNF as input. We translate every feature model to CNF using FeatureIDE [64] prior to the experiments. This transformation produces an equivalent CNF without introducing any artificial variables [54].

Experiment 1: Compilation to d-DNNF. In the first experiment, we measure the runtimes required to translate CNFs representing our subject systems to d-DNNF. Also, we examine the sizes of the resulting d-DNNF. Hereby, we evaluate each d-DNNF compiler, namely c2d, dSharp, and d4, on all 62 subject systems with a timeout of 30 minutes per feature model. The timeout of 30 minutes throughout the experiments is motivated by: (1) we expect that higher runtimes are typically unsuitable for interactive applications and continuous integration pipelines and (2) preliminary experiments showed that further increasing the timeout does not lead to substantial differences in the results. For each combination of compiler and subject system, we perform 50 repetitions. If not stated otherwise, we use the median over the 50 repetitions in plots in this evaluation. The compiled d-DNNFs are used for Experiment 2 and Experiment 3. The insights from Experiment 1 are used to answer RQ1.

Experiment 2: Cardinality of Features. In the second experiment, we measure the runtimes required to compute the cardinalities of all features in a given feature model. Hereby, we evaluate each d-DNNF tool combination (consisting of compiler and reasoner) and #SAT solver on each subject system. We set a timeout of 30 minutes for computing the cardinality of all features for a single feature model. For each measurement, we perform 50 repetitions. The insights from Experiment 2 are used to answer RQ2–RQ4. Note that some feature models [74] are only available as propositional

[19] https://www.cril.univ-artois.fr/KC/d4.html
[20] https://github.com/meelgroup/ganak/tree/18af6360fbb3e4899bc0a6135fd9167c08f10a19
[21] https://github.com/QuMuLab/dsharp/tree/5a44909788d077d62ce9a731f1ac453c556ef891
[22] https://github.com/SoftVarE-Group/d-dnnf-reasoner/tree/abfaa711bad9629d735906ab714919e12f777404
[23] https://mccompetition.org/index.html

Table 3. Classification Effect Sizes of Cohen's d Adapted from Sullivan and Feinn [89]

Cohen's d	Effect Size Classification
$0.0 \leq d < 0.2$	Very Small
$0.2 \leq d < 0.5$	Small
$0.5 \leq d < 0.8$	Medium
$0.8 \leq d < 1.3$	Large
$1.3 \leq d$	Very Large

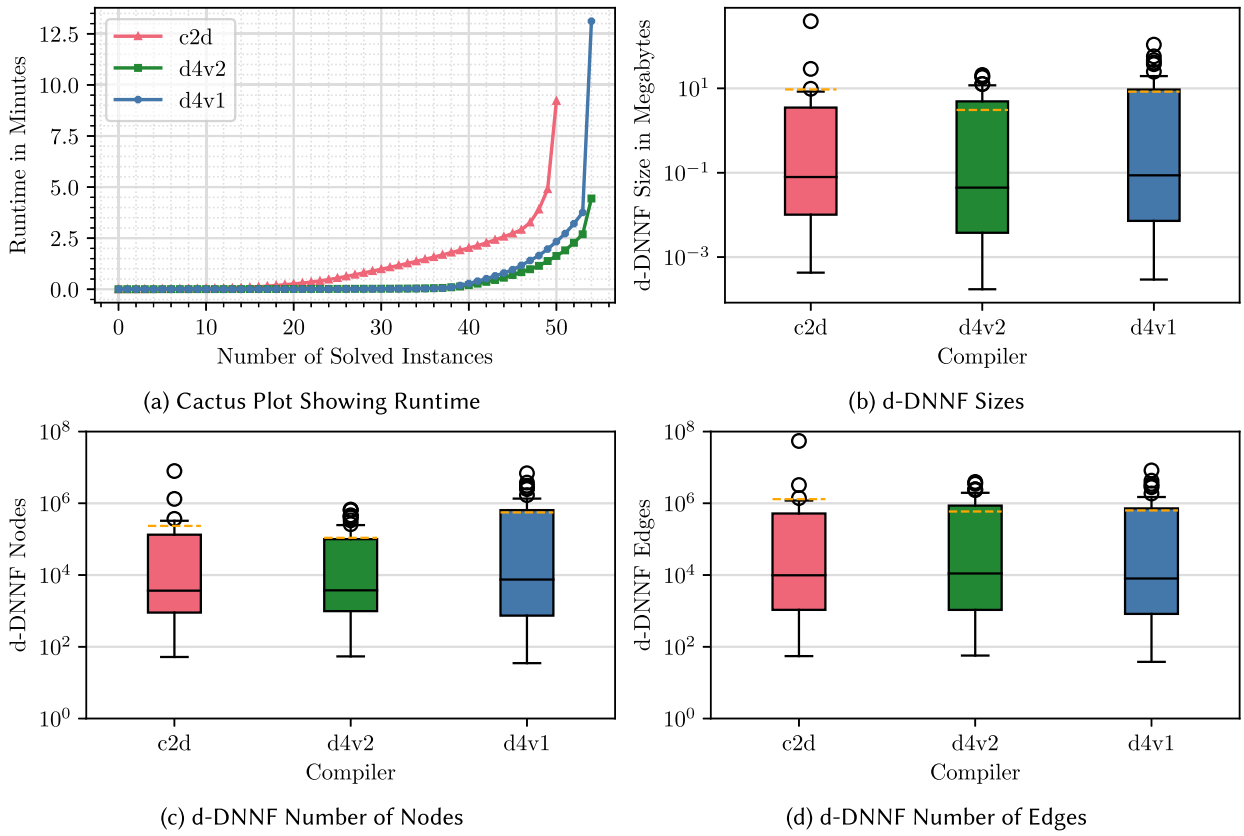
formulas in DIMACS format without further information on the semantics of the variables. Hence, we consider the cardinality for each variable here.

Experiment 3: Cardinality of Partial Configurations. In the third experiment, we measure the runtimes required to compute the cardinality of partial configurations for a given feature model. As we have no real queries available, we randomly generate 250 partial configurations for each subject system. We ensure that each partial configuration is satisfiable, as we assume that satisfiable configurations better reflect practical usage of feature-model counting based on the applications found in the literature [92]. To generate the partial configurations, we iteratively select or deselect a randomly chosen feature. If this results in an invalid configuration, we discard the current selection and continue. The 250 configurations are separated in 50 chunks with sizes of 2, 5, 10, 20, and 50 features (randomly included or excluded), respectively. For the ten feature models with fewer than 50 features, we skip chunks exceeding the number of features in the model. With the different configuration sizes, we aim to cover a wide range of analyses dependent on the cardinality of partial configurations [92]. As for Experiment 2, we evaluate each d-DNNF tool combination and #SAT solver on each subject system. We set a timeout of 30 minutes for computing the cardinality of all 250 partial configurations for a single feature model. For each measurement, we perform 50 repetitions. The insights from Experiment 3 are used to answer RQ2–RQ4.

Statistical Tests. We use a Wilcoxon signed-ranked test [100] to compare the performance of two tools on the dataset. For comparing multiple tools, we perform a Friedman test followed by a *post hoc* Conover test [19]. We use those tests as (1) we do not assume a normal distribution and (2) the runtimes are not independent of each other. Hereby, we always assume a significance level of $\alpha = 0.05$. To examine the effect sizes for significant results, we use Cohen's d [89]. For rating the effect size, we use the classification shown in Table 3 as suggested by Sullivan and Feinn [89]. For instance, we consider a value of $0.5 \leq d < 0.8$ as medium effect size.

Technical Setup. The entire empirical evaluation was performed on an Ubuntu 20.04.3 server with a 64-bit architecture. The processor is an Intel(R) Xeon(R) CPU E5-260v3 with 2.40 GHz clock rate. Overall, 256 GB of RAM were available. During the measurements, the machine was only used for our evaluation to reduce the impact of other processes. For each tool, the memory limit is set to 8 GB as we consider 8 GB to be a reasonable limit for hardware (e.g., notebooks or PCs) in practice. Furthermore, in preliminary experiments we found no major differences in measured runtimes if we further increased the memory limit. To measure runtimes, we used a Python-based framework which internally uses the module `timeit`.[24]

[24] https://docs.python.org/3/library/timeit.html



(a) Cactus Plot Showing Runtime

(b) d-DNNF Sizes

(c) d-DNNF Number of Nodes

(d) d-DNNF Number of Edges

Fig. 7. Results d-DNNF compilers.

6.3 Results

Experiment 1. Figure 7(a) shows the runtimes of the d-DNNF compilers for transforming the subject systems as cactus plot. The x-axis indicates the number of solved instances. The y-axis shows the cumulative runtime in seconds. In Experiment 2 and 3 (see below), we observed that `dSharp` produces faulty d-DNNFs and model counts. Hence, we exclude the compilation times of `dSharp` here as well. For 54 out of the 61 subject systems,[25] at least one d-DNNF compiler successfully compiled the CNF into d-DNNF within 30 minutes. `d4` (both versions) successfully compiled 54 feature models while `c2d` successfully compiled 52. For the 52 models successfully evaluated by both compilers, `d4` required 4.62 (14.3 for old version) seconds on average and `c2d` 69.5 seconds.

Figure 7(b) shows the sizes of the compiled d-DNNFs. For every boxplot, the dashed orange line indicates the average and the solid black line within the box indicates the median. The circles outside the boxes represent outliers. The majority (92.5%) of d-DNNFs require less than 10 MB of space. All but one[26] d-DNNF are smaller than 50 MB. The d-DNNFs produced by the fastest compiler (`d4`) require at most 20.1 MB (median 0.04 MB). Those d-DNNFs compiled by `d4` have 54–665,415 nodes and 57–3.94 $\times$ 10^6 edges. The fact that the number of edges is higher than the number of nodes indicates the reuse of d-DNNF subtrees. For instance, `d4` d-DNNFs have 3.4 times as many edges as nodes on average.

Experiment 2. All tools except `dSharp` and `sharpSAT` computed the same results (i.e., cardinalities) for each feature of all considered feature models. The results of `dSharp` (as black box) and `sharpSAT` differed from those results for 57.5% (31 affected models) and 14.8% (8) of the overall computed

[25] Failed for: four Linux variants, Embtoolkit, Freetz, and Automotive05.
[26] Automotive02, compiled by `c2d`, required 391 MB.

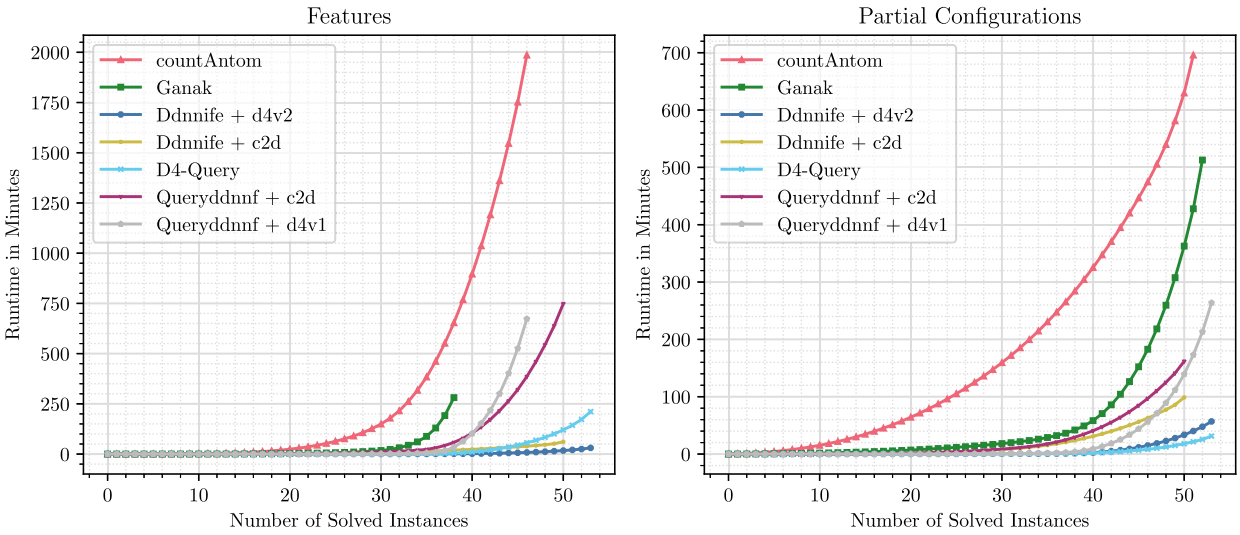


Fig. 8. Cactus plot showing runtime for offline + online phase.

cardinalities, respectively. `ddnnife` with d-DNNFs compiled by `dSharp` differed for 0.54%. Due to the inconsistent results based on `dSharp` and `sharpSAT`, we decided to exclude them from runtime comparisons for this and following experiments.

Figure 8 shows a cactus plot indicating the cumulative runtimes required to compute the cardinality of all features and partial configurations using d-DNNFs (offline + online phases) and repetitive #SAT calls. For each feature model successfully compiled to d-DNNF, at least one d-DNNF tool combination was able to successfully compute all feature cardinalities within 30 minutes. The #SAT solvers evaluated all feature cardinalities for only 47 feature models, failing for each feature model the d-DNNF compilers were not able to compile and seven additional ones. For feature models also successfully evaluated by a #SAT solver, applying d-DNNFs is overall 71.4 times faster. If we perform a linear extrapolation from the average time required to evaluate a variable,[27] `ddnnife` with `d4` is up-to ~6,400 (~15,388 without compilation) times faster than the fastest #SAT solver.

Figure 9 shows the relative runtimes of the d-DNNF-based tools compared to the performance of `ddnnife`. The second row shows only the online phases while the first row also includes compilation times. Note that `d4-query` has no online phase and the runtimes in the first and second row are equal. Within each box, the dashed line indicates the average and the solid line the median runtime. For `ddnnife` and `query-ddnnf`, the runtimes of the respective best performing compiler (`d4` and `c2d`) are included. Overall, `ddnnife` combined with `d4` is the fastest d-DNNF-based tool combination requiring 6.02 minutes for solving 54 feature models. `d4-query` requires 40 minutes for the same 54 feature models. This difference between `ddnnife` and `d4-query` is significant ($p = 5.3 \times 10^{-7}$) with a medium effect size ($d = 0.65$). `query-ddnnf` with `c2d` solves all cardinalities for only 53 models and requires 175 minutes for them. For every feature model requiring more than 0.1 seconds to evaluate with all three tool (combinations), `ddnnife` with `d4` requires the least runtime. Excluding compilation time, `ddnnife` requires 1.31 minutes for all 54 feature models while `query-ddnnf` requires 72.0 minutes for 53 models.

Experiment 3. All tools but `dSharp` computed the same cardinalities for every partial configuration and feature model. The results of `dSharp` (black box) and `ddnnife` with `dSharp` differed from those

[27] From sample-based tests, we observed that the runtimes are similar for the vast majority of variables. Hence, we consider 153 variables are valid sample for such an extrapolation.

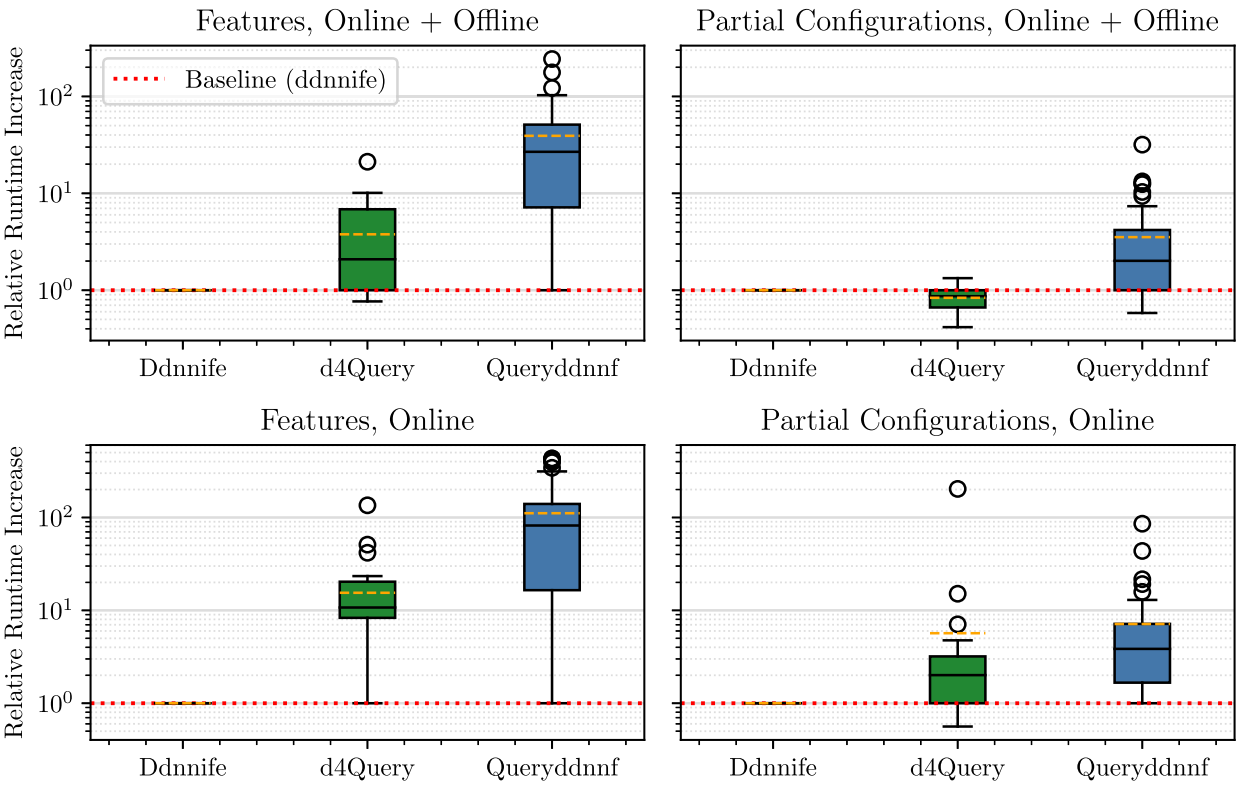


Fig. 9. Runtime comparison of d-DNNF-based tools (runtimes relative to `ddnnife`).

results in 41.83% (31 affected models) and 5.83% (31 affected models) cases, respectively. Their runtimes are excluded from the runtime comparisons as in Experiment 2.

The right side of Figure 8 shows the runtimes required to compute the cardinality of the 250 generated partial configurations with d-DNNFs and repetitive #SAT calls. `ddnnife` and `query-ddnnf` in combination with any compiler and `d4-query` are able to compute the cardinality for the 250 partial configurations within 30 minutes for every successfully compiled d-DNNF (overall 54). The fastest #SAT solver `Ganak` successfully evaluated 53. For those 53 models, the fastest runtimes amount to 3.87 minutes for d-DNNF and 51.8 minutes for #SAT.

The right side of Figure 9 shows the relative runtimes of the d-DNNF-based tools compared to `ddnnife` for computing the cardinality of partial configurations. For partial configurations, `query-ddnnf` requires the least overall runtime with 5.88 minutes. `ddnnife` and `query-ddnnf` with `d4` require 9.24 minutes and 50.7 minutes, respectively. Both, `d4-query` and `ddnnife` are significantly faster than `query-ddnnf`, with $p < 10^{-6}$, $d = 0.80$, and $p < 10^{-4}$, $d = 0.53$, respectively. The difference between `ddnnife` and `d4-query` is not significant ($p = 0.33$). Without considering compilation time, `ddnnife` requires 4.80 minutes and `query-ddnnf` 37.5 minutes.

6.4 Discussion

RQ1 How efficient are d-DNNF compilers when applied to industrial feature models regarding runtime and memory? Our results indicate that compilation to d-DNNF is promising for feature-model counting. Compiling to d-DNNF seems to only fail for feature models that are very hard for feature-model counting in general, as each #SAT failed for the same feature models. This observation matches the expectation as popular d-DNNF compilers and #SAT behave very similarly [24, 56, 68, 97]. In most environments, the memory usage of d-DNNFs should also cause no problems. The fastest compiler `d4` produces d-DNNFs requiring at most 20.1 MB. Overall, considering d-DNNF size

and compilation time, d4 is the most promising d-DNNF compiler for feature models. Nevertheless, since every considered d-DNNF compiler failed for seven of the 61 systems, there is potential and demand for improving the scalability of compilers for feature models. These performance results are in line with previous examinations on the performance of d-DNNF compilers [56, 90, 91]. Hence, our results suggest that the compilers behave similarly to other Boolean instances (e.g., formulas representing bayesian networks [56]) when compiling industrial feature models.

RQ2 Which tools and tool combinations compute correct cardinalities? `countAntom`, `Ganak`, `d4`, `c2d`, `ddnnife` (with `c2d` and `d4`), and `query-ddnnf` computed the same cardinalities for every single computation which we consider as a strong indicator for the correctness of these tools. In particular, due to more than 200k consistent results, it is very likely that the algorithms and optimizations presented in this submission and implemented in `ddnnife` yield correct results. Only on faulty d-DNNFs produced by `dSharp`, `ddnnife` computes a few differing cardinalities, which is very likely caused by `dSharp` providing incorrect d-DNNFs in those cases. Due to the incorrect results, we did not consider `sharpSAT` and `dSharp` for runtime comparisons.

As `sharpSAT` has been used for analyzing feature models in various publications [4, 35, 55, 57, 69, 74, 90], our results on its correctness may also have implications for those results. Also, the errors indicate the importance of validating model counting results as recognizing a faulty result is difficult without a reference value.

RQ3 How efficient is reusing d-DNNFs for computing cardinalities of features and partial configurations compared to repetitive #SAT calls? Reusing d-DNNFs for repetitive queries of feature-model counting is substantially faster than using state-of-the-art #SAT solvers for every considered feature model. Often, applying d-DNNFs is multiple orders of magnitude faster than using repetitive #SAT invocations. Overall, we argue the effort of compilation to d-DNNF is worth for feature-model counting and substantially more efficient than the current state-of-the-art of repetitive #SAT queries. As d-DNNFs for repetitive counting queries have not been empirically evaluated in any domain before, our results provide the first insights on the performance. The substantial improvements over repetitively invoking #SAT solvers suggest that reusing d-DNNFs may also be promising for other domains.

RQ4 Which d-DNNF-based tools perform best for computing cardinalities? `ddnnife` with `d4` and `d4-query` appear to be the most promising options for repetitive queries of feature model counting. Overall, the combination of `ddnnife` and `d4` is the fastest requiring 10.8 minutes for computing the cardinalities of features and partial configurations on all feature models followed by `d4-query` (46.9 minutes). For cardinality of features, `ddnnife` significantly outperforms `d4-query`. For the experiments on partial configurations, we have fewer queries and also more assumptions per query. While `ddnnife` is still fastest when a d-DNNF is already given, the performance advantages are smaller. Consequently, more queries are needed to amortize the compilation to d-DNNF when compared to `d4-query`. Thus, when the d-DNNF is not needed for further reuse and only few queries are computed `d4-query` can be advantageous. However, applications suggested in the literature [36, 53, 92] often demand a high number of partial configuration queries [91]. For instance, supporting the interactive configuration process by providing an indication on the impact of decisions on the configuration space may require up-to $O(|features|^2)$ queries [62, 91]. We argue that `ddnnife` is likely to amortize the compilation cost compared to `d4-query` for such applications. Still, it may be beneficial to further improve the performance of `ddnnife` for partial configurations. One option may be sorting queries to reduce the share of nodes adapted by subsequent queries. Furthermore, partial derivatives could be also applied for partial configurations if only one literal is flipped from a previous partial configuration.

6.5 Threats to Validity

Translating Feature Models to CNF. Each evaluated compiler and #SAT solver uses CNF as input format. To employ the tools, the feature models need to be translated to CNF. Two equivalent but syntactically different CNFs may result in significantly varying runtime for solvers and compilers [46, 54, 74]. Hence, changing the CNF translation may have an impact on the measured runtimes. However, analyzing multiple CNF translations is beyond the scope of this work. Another aspect to consider is that some CNF translations are equisatisfiable but do not preserve the number of satisfying assignments. Such a translation may lead to faulty cardinalities in our analyses. For each feature model given in a proprietary format, we use the transformation to CNF employed by FeatureIDE [64] which preserves equivalence and, thus, the number of satisfying assignments [54]. The remaining feature models (collected by Oh et al. [75]) were already specified in CNF and have been used in other empirical evaluations [44, 74, 75, 81].

Parameterization of Solvers and Compilers. All evaluated #SAT solvers and compilers provide parameters that potentially impact their runtime. It is possible that different parameters may significantly change the performance of evaluated tools. However, we expect that using default parameters reasonably represents the usage in practice. Parameter optimization for each solver and compiler is out of scope for this work.

Random Effects and Computational Bias. When measuring the runtime of a solver required to analyze a feature model, the runtimes may vary between measurements due to computational bias, random effects, or background processes. In particular, the performance of `c2d` is reliant on an initial decomposition tree which is heavily dependent on random decisions. To reduce the impact of such influence factors, we perform 50 repetitions for each measurement present the median if not stated otherwise. Also, while performing measurements no other major processes were executed. Furthermore, we employ established statistical tests to validate the significance of our conclusions.

Memory Limit. For each tool, we limited the memory usage to 8 GB due to expected requirements in practice and preliminary experiments (cf. Section 6.2 for more details). Still, changing the memory limit may have an impact on the performance of the different tools. However, repeating all experiments with multiple memory limits would have required vast computational resources and further increase the complexity of the evaluation.

External Validity—Selection of Queries. For evaluating the tools on computing cardinality of features, we consider the cardinality of all features. Considering fewer features may have an impact on the performance of the different tools. However, many analyses considered in the literature often require to analyze many or even all features [92]. In particular, the metric *homogeneity*, which has often been considered [18, 29, 30, 37, 92], by definition requires to compute the cardinality of all features. Also, `ddnnife` amortizes the initial cost of compiling after few queries compared to #SAT solvers.

For cardinality of partial configurations, we use randomly generated configurations of different sizes. It is possible that the random decisions do not fully reflect the queries required for specific analyses in practice. However, the different analyses dependent on partial configurations require multiple settings with varying number of (de-)selected features [53, 74, 92]. With our evaluation, we aim to not be limited to single analyses. Due to the heterogeneity of configurations in different analyses and real partial configurations typically not being available, we consider random configurations of various sizes as reasonable decision for our experiments.

External Validity—Tools. The measured performance of the evaluated tools cannot be necessarily transferred to other tools. However, to the best of our knowledge, we evaluated all publicly available

d-DNNF compilers and reasoners. For #SAT solvers, we selected the two best performing #SAT solvers for feature-model counting according to our previous experiments [93] and the winning #SAT solver of the model counting competition 2020. Hence, we expect our results to reasonably represent the performance of both approaches.

External Validity—Subject Systems. Our conclusions do not necessarily hold for other real-world feature models. However, we evaluated feature models from a variety of domains, namely systems software, automotive, financial services, and multiple other domains. Furthermore, we covered a wide range of features (11–62,482), clauses (1–350,221), number of valid configurations (1,024–10^{1534}), and induced runtimes for solving #SAT (between few milliseconds and hitting timeouts of 24 hours [93]).

7 Related Work

In this section, we discuss work that (a) performs model counting with d-DNNFs, (b) uses d-DNNFs in the context of feature models, (c) proposes alternative approaches to allow feature-model counting, (d) applies knowledge compilation in feature-model analysis, or (e) presents other knowledge-compilation artifacts for which model counting is tractable.

Model Counting with d-DNNFs. Each considered d-DNNF compiler, namely `c2d` [24], `dSharp` [68], and `d4` [56], is able to compute the number of satisfying assignments of the input. However, neither compiler allows reusing a given d-DNNF to compute cardinalities of features or partial configurations. While `d4` (given a CNF) and `query-ddnnf` support computing multiple queries, those functionalities have not been considered in a scientific publication and have not been evaluated before. Generally, there has not been any empirical evaluation on reusing d-DNNFs for counting.

d-DNNF Exploitation in Feature-Model Analysis. Sharma et al. [86] use d-DNNFs for uniform random sampling with their tool *KUS*. Internally, *KUS* depends on counting on a given d-DNNF. While that counting procedure could be extracted, it does not support applying assumptions (here: including/excluding features) and, thus, cannot be used for both query types we consider. Bourhis et al. [15] employ d-DNNFs for multiple different feature-model analyses including counting. However, for counting, they only consider single computations for the entire feature model while we target repetitive queries for features and partial configurations. In previous publications, we [93] and Kübler et al. [53] evaluate d-DNNF compilers for the task of computing the number of valid configurations. However, in both publications the compilers are just used as black-box #SAT solvers, essentially compiling a new d-DNNF for every single computation, and do not reuse the compiled d-DNNF. Furthermore, no repeated queries (e.g., for cardinality of features) are performed.

Solutions for Feature-Model Counting. In the above mentioned works, Kübler et al. [53] and we [93] compute the cardinalities of industrial feature models. Both works are limited to one query per feature model and only use the solvers as black-box, whereas we reuse the compiled the d-DNNFs to reduce the effort of repetitive queries.

Pohl et al. [83] compare the runtimes of several solver types, namely BDD, SAT, and CSP, on different feature-model analyses, including feature-model counting. The results indicate that BDDs scale substantially better than SAT and CSP for computing the number of valid configurations. However, their dataset contains only comparatively small feature models with less than 300 features, whereas our benchmark contains feature models with several ten-thousands features. In previous work, we found that compiling to BDD scales for only very few industrial feature models and considerably worse than d-DNNFs [39, 90].

Other work proposes *ad hoc* solutions for feature model counting targeting feature models with no [38] or very few cross-tree constraints [30]. While those perform well for the feature

models with no or very few cross-tree constraints, recent work indicates that industrial feature models tend to have very large number of cross-tree constraints (e.g., up-to 15k in our dataset) [39, 48, 74, 93].

Uniform-Random Sampling. Several approaches for uniform random sampling internally depend on counting [4, 69, 75, 86]. For instance, *Smarch* [74] builds a uniform sample by repetitively invoking `sharpSAT`. *Spur* [4] adapts `sharpSAT` to cache values of computation nodes to accelerate deriving valid configurations. However, like for *KUS* (as we described in *d-DNNF Exploitation in Feature-Model Analysis*), it is not possible to apply assumptions for *Spur*. Hence, computing the cardinality of features or partial configurations is also not possible without major conceptual adaptations.

Knowledge Compilation in Feature-Model Analysis. BDDs are often applied for feature-model analyses [1, 3, 10, 67, 73, 93], including some proposals to use BDDs for feature-model counting [39, 42, 65, 83, 93]. However, BDDs fail to scale for many industrial feature models [39, 74, 93, 94]. Krieter et al. [51, 52] employ modal implication graphs to accelerate decision propagation during product configuration. However, modal implication graphs are not tractable for feature-model counting.

Model Counting with Other Knowledge-Compilation Artifacts. Model counting is tractable for several knowledge-compilation artifacts (i.e., has a polynomial time complexity with respect to the size of artifacts). According to the knowledge compilation map of Darwiche and Marquis [26], BDDs, d-DNNFs, and MODs (short for models which is a DNF that satisfies smoothness and determinism) allow polynomial-time counting. In other work, **sentential decision diagrams (SDD)** [77] and **extended affine decision trees (EADT)** [49] have been found tractable for model counting. While all these formats allow counting in theory, we compared the performance of BDD, SDD, EADT, and d-DNNF compilers for the task of feature-model counting in recent work [90] and found d-DNNF compilers to be the most efficient.

Incremental Solving. An incremental solver performs multiple queries on highly similar formulas while keeping information between those queries to accelerate later invocations [28]. For regular SAT, several solvers were proposed with promising results for many use-cases [27, 28, 70]. Typically, those solvers are adaptations of popular SAT solvers based on Davis-Putnam-Logemann-Loveland [33] or conflict-driven clause learning [61]. While such incremental SAT solvers have been used for years, there have not been such adaptations for #SAT.

8 Conclusion

Many analyses based on feature-model counting require various similar counting queries to compute the number of valid configurations that contain a certain feature or conform to a partial configuration. We are the first to realize reusing d-DNNFs to reduce the effort of repetitive counting queries. Our evaluation shows that reusing d-DNNFs substantially reduces the runtime compared to the state-of-the-art of repetitively invoking #SAT solvers. We found that compilation to d-DNNF scales for every feature model for which state-of-the-art #SAT solvers are able to compute a result. Applying d-DNNFs is faster than #SAT for every considered feature model, that at least one tool was able to analyze, reducing runtimes by more than 99.9% in the best case including compilation time. Further, our tool `ddnnife` combined with the compiler `d4` was the overall fastest tools for the considered problems, namely cardinality of features and cardinality of partial configurations. We therefore recommend reusing d-DNNFs with `ddnnife` when many queries of feature-model counting are required. When only few queries are required and the d-DNNF is not further reused, using `d4-query` can yield benefits over `ddnnife`.

References

[1] Mathieu Acher, Philippe Collet, Philippe Lahire, and Robert B. France. 2011. Slicing feature models. In *Proceedings of the International Conference on Automated Software Engineering (ASE '11)*. IEEE, 424–427.

[2] Mathieu Acher, Philippe Collet, Philippe Lahire, and Robert B. France. 2013. Familiar: A domain-specific language for large scale management of feature models. *Science of Computer Programming (SCP)* 78, 6 (2013), 657–681.

[3] Mathieu Acher, Patrick Heymans, Philippe Collet, Clément Quinton, Philippe Lahire, and Philippe Merle. 2012. Feature model differences. In *Proceedings of the International Conference on Advanced Information Systems Engineering (CAiSE '12)*. Springer, 629–645.

[4] Dimitris Achlioptas, Zayd S. Hammoudeh, and Panos Theodoropoulos. 2018. Fast sampling of perfectly uniform satisfying assignments. In *Proceedings of the International Conference on Theory and Applications of Satisfiability Testing (SAT '18)*. Springer, 135–147.

[5] Carlos Ansótegui, María L. Bonet, and Jordi Levy. 2009. On the structure of industrial SAT instances. In *Proceedings of the International Conference on Principles and Practice of Constraint Programming (CP '09)*. Springer, 127–141.

[6] Ebrahim Bagheri and Dragan Gasevic. 2011. Assessing the maintainability of software product line feature models using structural metrics. *Software Quality Journal (SQJ)* 19, 3 (2011), 579–612.

[7] Don Batory. 2005. Feature models, grammars, and propositional formulas. In *Proceedings of the International Systems and Software Product Line Conference (SPLC '05)*. Springer, 7–20.

[8] Roberto J. Bayardo Jr. and Joseph D. Pehoushek. 2000. Counting models using connected components. In *Proceedings of the Conference on Artificial Intelligence (AAAI '00)*. AAAI Press, 157–162.

[9] David Benavides, Sergio Segura, and Antonio Ruiz-Cortés. 2010. Automated analysis of feature models 20 years later: A literature review. *Information Systems* 35, 6 (2010), 615–708.

[10] David Benavides, Sergio Segura, Pablo Trinidad, and Antonio Ruiz-Cortés. 2007. FAMA: Tooling a framework for the automated analysis of feature models. In *Proceedings of the International Workshop on Variability Modelling of Software-Intensive Systems (VaMoS)*. Technical Report 2007-01, Lero, 129–134.

[11] David Benavides, Pablo Trinidad, and Antonio Ruiz-Cortés. 2005. Using constraint programming to reason on feature models. In *Proceedings of the International Conference on Software Engineering and Knowledge Engineering (SEKE '05)*, 677–682.

[12] Thorsten Berger, Divya Nair, Ralf Rublack, Joanne M. Atlee, Krzysztof Czarnecki, and Andrzej Wąsowski. 2014. Three cases of feature-based variability modeling in industry. In *Proceedings of the International Conference on Model Driven Engineering Languages and Systems (MODELS '14)*. Springer, 302–319.

[13] Thorsten Berger, Ralf Rublack, Divya Nair, Joanne M. Atlee, Martin Becker, Krzysztof Czarnecki, and Andrzej Wąsowski. 2013. A survey of variability modeling in industrial practice. In *Proceedings of the International Workshop on Variability Modelling of Software-Intensive Systems (VaMoS '13)*. ACM, 7:1–7:8.

[14] Armin Biere. 2008. PicoSAT essentials. *Journal on Satisfiability, Boolean Modeling and Computation* 4 (2008), 75–97.

[15] Pierre Bourhis, Laurence Duchien, Jérémie Dusart, Emmanuel Lonca, Pierre Marquis, and Clément Quinton. 2023. *Reasoning on Feature Models: Compilation-Based vs. Direct Approaches*. Technical Report. Cornell University Library.

[16] Jan Burchard, Tobias Schubert, and Bernd Becker. 2015. Laissez-faire caching for parallel #SAT solving. In *Proceedings of the International Conference on Theory and Applications of Satisfiability Testing (SAT)*. Springer, 46–61.

[17] Sheng Chen and Martin Erwig. 2011. Optimizing the product derivation process. In *Proceedings of the International Systems and Software Product Line Conference (SPLC '11)*. IEEE, 35–44.

[18] Paul C. Clements, John D. McGregor, and Sholom G. Cohen. 2005. *The Structured Intuitive Model for Product Line Economics (SIMPLE)*. Technical Report. Carnegie Mellon University.

[19] William J. Conover and Ronald L. Iman. 1981. Rank transformations as a bridge between parametric and nonparametric statistics. *The American Statistician* 35, 3 (1981), 124–129.

[20] Krzysztof Czarnecki and Chang H. P. Kim. 2005. Cardinality-based feature modeling and constraints: A progress report. In *Proceedings of the International Workshop on Software Factories (SF '05)*, 16–20.

[21] Krzysztof Czarnecki and Andrzej Wąsowski. 2007. Feature diagrams and logics: There and back again. In *Proceedings of the International Systems and Software Product Line Conference (SPLC '07)*. IEEE, 23–34.

[22] Adnan Darwiche. 2001. Decomposable negation normal form. *Journal of the ACM* 48, 4 (2001), 608–647.

[23] Adnan Darwiche. 2001. On the tractable counting of theory models and its application to truth maintenance and belief revision. *Journal of Applied Non-Classical Logics* 11, 1–2 (2001), 11–34.

[24] Adnan Darwiche. 2002. A compiler for deterministic, decomposable negation normal form. In *Proceedings of the Conference on Artificial Intelligence (AAAI '02)*. AAAI Press, 627–634.

[25] Adnan Darwiche. 2004. New advances in compiling CNF to decomposable negation normal form. In *Proceedings of the European Conference on Artificial Intelligence*. IOS Press, 318–322.

[26] Adnan Darwiche and Pierre Marquis. 2002. A knowledge compilation map. *Journal of Artificial Intelligence Research (JAIR)* 17, 1 (2002), 229–264.

[27] Niklas Eén and Niklas Sörensson. 2003. Temporal induction by incremental SAT solving. *Electronic Notes in Theoretical Computer Science (ENTCS)* 89, 4 (2003), 543–560.
[28] Katalin Fazekas, Armin Biere, and Christoph Scholl. 2019. Incremental inprocessing in SAT solving. In *Proceedings of the International Conference on Theory and Applications of Satisfiability Testing (SAT '19)*. Mikoláš Janota and Inês Lynce (Eds.), Springer, 136–154.
[29] David Fernandez-Amoros, Ruben Heradio Gil, and Jose Cerrada Somolinos. 2009. Inferring information from feature diagrams to product line economic models. In *Proceedings of the International Systems and Software Product Line Conference (SPLC '09)*. Springer, 41–50.
[30] David Fernández-Amorós, Ruben Heradio, José A. Cerrada, and Carlos Cerrada. 2014. A scalable approach to exact model and commonality counting for extended feature models. *IEEE Transactions on Software Engineering (TSE)* 40, 9 (2014), 895–910.
[31] Johannes K. Fichte, Markus Hecher, and Florim Hamiti. 2021. The model counting competition 2020. *ACM Journal of Experimental Algorithmics (JEA)* 26 (2021), Article 13, 26 pages.
[32] Claudia Fritsch, Richard Abt, and Burkhardt Renz. 2020. The benefits of a feature model in banking. In *Proceedings of the International Systems and Software Product Line Conference (SPLC '20)*. ACM, Article 9, 11 pages.
[33] Harald Ganzinger, George Hagen, Robert Nieuwenhuis, Albert Oliveras, and Cesare Tinelli. 2004. DPLL (T): Fast decision procedures. In *Proceedings of the International Conference on Computer Aided Verification (CAV '04)*. Springer, 175–188.
[34] Marc Hentze, Tobias Pett, Chico Sundermann, Sebastian Krieter, Thomas Thüm, and Ina Schaefer. 2022. Generic solution-space sampling for multi-domain product lines. In *Proceedings of the International Conference on Generative Programming: Concepts & Experiences (GPCE '22)*. ACM, 135–147.
[35] Marc Hentze, Chico Sundermann, Thomas Thüm, and Ina Schaefer. 2022. Quantifying the variability mismatch between problem and solution space. In *Proceedings of the International Conference on Model Driven Engineering Languages and Systems (MODELS '22)*. IEEE, 322–333.
[36] Ruben Heradio, David Fernández-Amorós, José A. Cerrada, and Ismael Abad. 2013. A literature review on feature diagram product counting and its usage in software product line economic models. *International Journal of Software Engineering and Knowledge Engineering (IJSEKE)* 23, 08 (2013), 1177–1204.
[37] Ruben Heradio, David Fernández-Amorós, Christoph Mayr-Dorn, and Alexander Egyed. 2019. Supporting the statistical analysis of variability models. In *Proceedings of the International Conference on Software Engineering (ICSE '19)*. IEEE, 843–853.
[38] Rubén Heradio-Gil, David Fernández-Amorós, José Antonio Cerrada, and Carlos Cerrada. 2011. Supporting commonality-based analysis of software product lines. *IET Software* 5, 6 (2011), 496–509.
[39] Tobias Heß, Chico Sundermann, and Thomas Thüm. 2021. On the scalability of building binary decision diagrams for current feature models. In *Proceedings of the International Systems and Software Product Line Conference (SPLC '21)*. ACM, 131–135.
[40] Jinbo Huang and Adnan Darwiche. 2005. DPLL with a trace: From SAT to knowledge compilation. In *Proceedings of the International Joint Conference on Artificial Intelligence (IJCAI)*, Vol. 5. Professional Book Center, 156–162.
[41] Ayelet Israeli and Dror G. Feitelson. 2010. The Linux kernel as a case study in software evolution. *Journal of Systems and Software (JSS)* 83, 3 (2010), 485–501.
[42] Martin F. Johansen, Øystein Haugen, Franck Fleurey, Erik Carlson, Jan Endresen, and Tormod Wien. 2012. A technique for agile and automatic interaction testing for product lines. In *Proceedings of the International Conference on Testing Software and Systems (ICTSS)*. Springer, 39–54.
[43] David S. Johnson. 1992. The NP-completeness column: An ongoing guide. *Journal of Algorithms* 13, 3 (1992), 502–524.
[44] Christian Kaltenecker, Alexander Grebhahn, Norbert Siegmund, Jianmei Guo, and Sven Apel. 2019. Distance-based sampling of software configuration spaces. In *Proceedings of the International Conference on Software Engineering (ICSE '19)*. IEEE, 1084–1094.
[45] Vladimir Klebanov, Norbert Manthey, and Christian Muise. 2013. SAT-based analysis and quantification of information flow in programs. In *Proceedings of the International Conference on Quantitative Evaluation of Systems (QEST '13)*. Springer, 177–192.
[46] Will Klieber and Gihwon Kwon. 2007. Efficient CNF encoding for selecting 1 from n objects. In *Proceedings of the International Workshop on Constraints in Formal Verification (CFV '07)*. Springer, 17.
[47] Alexander Knüppel, Thomas Thüm, Stephan Mennicke, Jens Meinicke, and Ina Schaefer. 2017. Is there a mismatch between real-world feature models and product-line research?. In *Proceedings of the European Software Engineering Conference/Foundations of Software Engineering (ESEC/FSE '17)*. ACM, 291–302.
[48] Alexander Knüppel, Thomas Thüm, Stephan Mennicke, Jens Meinicke, and Ina Schaefer. 2018. Is there a mismatch between real-world feature models and product-line research?. In *Proceedings of the Software Engineering (SE '18)*.

Matthias Tichy, Eric Bodden, Marco Kuhrmann, Stefan Wagner, and Jan-Philipp Steghöfer (Eds.), Gesellschaft für Informatik, 53–54.

[49] Frédéric Koriche, Jean-Marie Lagniez, Pierre Marquis, and Samuel Thomas. 2013. Knowledge compilation for model counting: Affine decision trees. In *Proceedings of the International Joint Conference on Artificial Intelligence (IJCAI '13)*. AAAI Press.

[50] Matthias Kowal, Sofia Ananieva, and Thomas Thüm. 2016. Explaining anomalies in feature models. In *Proceedings of the International Conference on Generative Programming: Concepts & Experiences (GPCE '16)*. ACM, 132–143.

[51] Sebastian Krieter, Rahel Arens, Michael Nieke, Chico Sundermann, Tobias Heß, Thomas Thüm, and Christoph Seidl. 2021. Incremental construction of modal implication graphs for evolving feature models. In *Proceedings of the International Systems and Software Product Line Conference (SPLC '21)*. ACM, 64–74.

[52] Sebastian Krieter, Thomas Thüm, Sandro Schulze, Reimar Schröter, and Gunter Saake. 2018. Propagating configuration decisions with modal implication graphs. In *Proceedings of the International Conference on Software Engineering (ICSE '18)*. ACM, 898–909.

[53] Andreas Kübler, Christoph Zengler, and Wolfgang Küchlin. 2010. Model Counting in Product Configuration. In *Proceedings of the International Workshop on Logics for Component Configuration (LoCoCo '10)*. Open Publishing Association, 44–53.

[54] Elias Kuiter, Sebastian Krieter, Chico Sundermann, Thomas Thüm, and Gunter Saake. 2022. Tseitin or not Tseitin? The impact of CNF transformations on feature-model analyses. In *Proceedings of the International Conference on Automated Software Engineering (ASE '22)*. ACM, 110:1–110:13.

[55] Elias Kuiter, Sebastian Krieter, Chico Sundermann, Thomas Thüm, and Gunter Saake. 2023. Tseitin or not Tseitin? The impact of CNF transformations on feature-model analyses. In *Proceedings of the Software Engineering (SE '23)*, Vol. P-322. Gesellschaft für Informatik, 83–84.

[56] Jean-Marie Lagniez and Pierre Marquis. 2017. An improved decision-DNNF compiler. In *Proceedings of the International Joint Conference on Artificial Intelligence (IJCAI '17)*. International Joint Conferences on Artificial Intelligence, 667–673.

[57] Jia H. Liang, Vijay Ganesh, Krzysztof Czarnecki, and Venkatesh Raman. 2015. SAT-based analysis of large real-world feature models is easy. In *Proceedings of the International Systems and Software Product Line Conference (SPLC '15)*. Springer, 91–100.

[58] Michael Lienhardt, Ferruccio Damiani, Einer Broch Johnsen, and Jacopo Mauro. 2020. Lazy product discovery in huge configuration spaces. In *Proceedings of the International Conference on Software Engineering (ICSE '20)*. ACM, 1509–1521.

[59] Jackson A. Prado Lima, Willian D. F. Mendonça, Silvia R. Vergilio, and Wesley K. G. Assunção. 2020. Learning-based prioritization of test cases in continuous integration of highly-configurable software. In *Proceedings of the International Systems and Software Product Line Conference (SPLC '20)*. ACM, Article 31, 11 pages.

[60] Rafael Lotufo, Steven She, Thorsten Berger, Krzysztof Czarnecki, and Andrzej Wąsowski. 2010. Evolution of the Linux kernel variability model. In *Proceedings of the International Systems and Software Product Line Conference (SPLC '10)*. Springer, 136–150.

[61] Joao Marques-Silva, Inês Lynce, and Sharad Malik. 2009. Conflict-driven clause learning SAT solvers. In *Handbook of Satisfiability*. IOS Press, 131–153.

[62] Raúl Mazo, Cosmin Dumitrescu, Camille Salinesi, and Daniel Diaz. 2014. Recommendation heuristics for improving product line configuration processes. In *Recommendation Systems in Software Engineering*. Springer, 511–537.

[63] John McGregor. 2010. Testing a software product line. In *Testing Techniques in Software Engineering*. Springer, 104–140.

[64] Jens Meinicke, Thomas Thüm, Reimar Schröter, Fabian Benduhn, Thomas Leich, and Gunter Saake. 2017. *Mastering Software Variability With FeatureIDE*. Springer.

[65] Marcílio Mendonça, Moises Branco, and Donald Cowan. 2009. S.P.L.O.T.: Software product lines online tools. In *Proceedings of the Conference on Object-Oriented Programming, Systems, Languages and Applications (OOPSLA '65)*. ACM, 761–762.

[66] Marcílio Mendonça, Andrzej Wąsowski, and Krzysztof Czarnecki. 2009. SAT-based analysis of feature models is easy. In *Proceedings of the International Systems and Software Product Line Conference (SPLC '09)*. Software Engineering Institute, 231–240.

[67] Marcílio Mendonça, Andrzej Wąsowski, Krzysztof Czarnecki, and Donald Cowan. 2008. Efficient compilation techniques for large scale feature models. In *Proceedings of the International Conference on Generative Programming and Component Engineering (GPCE '08)*. ACM, 13–22.

[68] Christian Muise, Sheila A. McIlraith, J. Christopher Beck, and Eric I. Hsu. 2012. Dsharp: Fast d-DNNF compilation with sharpSAT. In *Proceedings of the Advances in Artificial Intelligence*. Leila Kosseim and Diana Inkpen (Eds.), Springer, 356–361.

[69] Daniel-Jesus Munoz, Jeho Oh, Mónica Pinto, Lidia Fuentes, and Don Batory. 2019. Uniform random sampling product configurations of feature models that have numerical features. In *Proceedings of the International Systems and Software Product Line Conference (SPLC '19)*. ACM, 289–301.

[70] Alexander Nadel, Vadim Ryvchin, and Ofer Strichman. 2014. Ultimately incremental SAT. In *Proceedings of the International Conference on Theory and Applications of Satisfiability Testing (SAT '14)*. Springer, 206–218.

[71] Michael Nieke, Jacopo Mauro, Christoph Seidl, Thomas Thüm, Ingrid Chieh Yu, and Felix Franzke. 2018. Anomaly analyses for feature-model evolution. In *Proceedings of the International Conference on Generative Programming: Concepts & Experiences (GPCE '18)*. ACM, 188–201.

[72] Michael Nieke, Gabriela Sampaio, Thomas Thüm, Christoph Seidl, Leopoldo Teixeira, and Ina Schaefer. 2022. Guiding the evolution of product-line configurations. *Software and Systems Modeling (SoSyM)* 21 (2022), 225–247.

[73] Jeho Oh, Don Batory, Margaret Myers, and Norbert Siegmund. 2016. *Finding Product Line Configurations With High Performance by Random Sampling*. Technical Report. University of Texas at Austin, Department of Computer Science.

[74] Jeho Oh, Paul Gazzillo, Don Batory, Marijn Heule, and Maggie Myers. 2019. *Uniform Sampling From Kconfig Feature Models*. Technical Report TR-19-02. University of Texas at Austin, Department of Computer Science.

[75] Jeho Oh, Paul Gazzillo, Don Batory, Marijn Heule, and Margaret Myers. 2020. *Scalable Uniform Sampling for Real-World Software Product Lines*. Technical Report TR-20-01. University of Texas at Austin, Department of Computer Science.

[76] Umut Oztok and Adnan Darwiche. 2014. On compiling CNF Into decision-DNNF. In *Proceedings of the International Conference on Principles and Practice of Constraint Programming (CP '14)*. Springer, 42–57.

[77] Umut Oztok and Adnan Darwiche. 2015. A top-down compiler for sentential decision diagrams. In *Proceedings of the International Joint Conference on Artificial Intelligence (IJCAI '15)*. AAAI Press, 3141–3148.

[78] Tobias Pett, Sebastian Krieter, Tobias Runge, Thomas Thüm, Malte Lochau, and Ina Schaefer. 2021. Stability of product-line sampling in continuous integration. In *Proceedings of the International Working Conference on Variability Modelling of Software-Intensive Systems (VaMoS '21)*. ACM, Article 18, 9 pages.

[79] Tobias Pett, Sebastian Krieter, Thomas Thüm, Malte Lochau, and Ina Schaefer. 2021. AutoSMP: An evaluation platform for sampling algorithms. In *Proceedings of the International Systems and Software Product Line Conference (SPLC '21)*. ACM, 41–44.

[80] Tobias Pett, Thomas Thüm, Tobias Runge, Sebastian Krieter, Malte Lochau, and Ina Schaefer. 2019. Product sampling for product lines: The scalability challenge. In *Proceedings of the International Systems and Software Product Line Conference (SPLC '19)*. ACM, 78–83.

[81] Quentin Plazar, Mathieu Acher, Gilles Perrouin, Xavier Devroey, and Maxime Cordy. 2019. Uniform sampling of SAT solutions for configurable systems: Are we there yet?. In *Proceedings of the International Conference on Software Testing, Verification and Validation (ICST '19)*. IEEE, 240–251.

[82] Klaus Pohl, Günter Böckle, and Frank J. van der Linden. 2005. *Software Product Line Engineering: Foundations, Principles and Techniques*. Springer.

[83] Richard Pohl, Kim Lauenroth, and Klaus Pohl. 2011. A performance comparison of contemporary algorithmic approaches for automated analysis operations on feature models. In *Proceedings of the International Conference on Automated Software Engineering (ASE '11)*. IEEE, 313–322.

[84] Steven D. Prestwich. 2009. CNF encodings. *Handbook of Satisfiability* 185 (2009), 75–97.

[85] Tian Sang, Fahiem Bacchus, Paul Beame, Henry A. Kautz, and Toniann Pitassi. 2004. Combining component caching and clause learning for effective model counting. In *Proceedings of the International Conference on Theory and Applications of Satisfiability Testing (SAT '04)*. Springer, 20–28.

[86] Shubham Sharma, Rahul Gupta, Subhajit Roy, and Kuldeep S. Meel. 2018. Knowledge compilation meets uniform sampling. In *Proceedings of the International Conference on Logic for Programming, Artificial Intelligence, and Reasoning*. EasyChair, 620–636.

[87] Shubham Sharma, Subhajit Roy, Mate Soos, and Kuldeep S. Meel. 2019. GANAK: A scalable probabilistic exact model counter. In *Proceedings of the International Joint Conference on Artificial Intelligence (IJCAI '19)*, Vol. 19. AAAI Press, 1169–1176.

[88] Joshua Sprey, Chico Sundermann, Sebastian Krieter, Michael Nieke, Jacopo Mauro, Thomas Thüm, and Ina Schaefer. 2020. SMT-based variability analyses in featureIDE. In *Proceedings of the International Working Conference on Variability Modelling of Software-Intensive Systems (VaMoS '20)*. ACM, Article 6, 9 pages.

[89] Gail M. Sullivan and Richard Feinn. 2012. Using effect size—or why the P value is not enough. *Journal of Graduate Medical Education* 4, 3 (2012), 279–282.

[90] Chico Sundermann, Tobias Heß, Michael Nieke, Paul M. Bittner, Jeffrey M. Young, Thomas Thüm, and Ina Schaefer. 2023. Evaluating state-of-the-art #SAT solvers on industrial configuration spaces. *Empirical Software Engineering (EMSE)* 28, 29 (2023), 38.

[91] Chico Sundermann, Elias Kuiter, Tobias Heß, Heiko Raab, Sebastian Krieter, and Thomas Thüm. 2023. On the benefits of knowledge compilation for feature-model analyses. *Annals of Mathematics and Artificial Intelligence (AMAI)* (2023).
[92] Chico Sundermann, Michael Nieke, Paul M. Bittner, Tobias Heß, Thomas Thüm, and Ina Schaefer. 2021. Applications of #SAT solvers on feature models. In *Proceedings of the International Working Conference on Variability Modelling of Software-Intensive Systems (VaMoS '21)*. ACM, Article 12, 10 pages.
[93] Chico Sundermann, Thomas Thüm, and Ina Schaefer. 2020. Evaluating #SAT solvers on industrial feature models. In *Proceedings of the International Working Conference on Variability Modelling of Software-Intensive Systems (VaMoS '20)*. ACM, Article 3, 9 pages.
[94] Thomas Thüm. 2020. A BDD for Linux? The knowledge compilation challenge for variability. In *Proceedings of the International Systems and Software Product Line Conference (SPLC '20)*. ACM, Article 16, 6 pages.
[95] Thomas Thüm, Sven Apel, Christian Kästner, Ina Schaefer, and Gunter Saake. 2014. A classification and survey of analysis strategies for software product lines. *ACM Computing Surveys (CSUR)* 47, 1 (2014), 6:1–6:45.
[96] Thomas Thüm, Don Batory, and Christian Kästner. 2009. Reasoning about edits to feature models. In *Proceedings of the International Conference on Software Engineering (ICSE '09)*. IEEE, 254–264.
[97] Marc Thurley. 2006. sharpSAT - Counting models with advanced component caching and implicit BCP. In *Proceedings of the International Conference on Theory and Applications of Satisfiability Testing (SAT '06)*. Springer, 424–429.
[98] Leslie G. Valiant. 1979. The complexity of enumeration and reliability problems. *SIAM Journal on Computing* 8, 3 (1979), 410–421.
[99] Frank J. van der Linden, Klaus Schmid, and Eelco Rommes. 2007. *Software Product Lines in Action: The Best Industrial Practice in Product Line Engineering*. Springer.
[100] R. F. Woolson. 2008. *Wilcoxon Signed-Rank Test*. John Wiley & Sons, 1–3 pages.
[101] Wei Zhang, Haiyan Zhao, and Hong Mei. 2004. A propositional logic-based method for verification of feature models. In *Proceedings of the International Conference on Formal Engineering Methods (ICFEM)*. Springer, 115–130.

Received 24 March 2023; revised 4 March 2024; accepted 21 June 2024

C.7 ASE'24: Efficient Slicing of Feature Models via Projected d-DNNF Compilation

[C3] Chico Sundermann, Jacob Loth, and Thomas Thüm. "Efficient Slicing of Feature Models via Projected d-DNNF Compilation". In: *Proc. Int'l Conf. on Automated Software Engineering (ASE)*. ACM, 2024, pp. 1332–1344. DOI: 10.1145/3691620.3695594

Published under Creative Commons Attribution 4.0 International (CC-BY), see https://creativecommons.org/licenses/by/4.0/.



Efficient Slicing of Feature Models via Projected d-DNNF Compilation

Chico Sundermann
University of Ulm
Ulm, Germany

Jacob Loth
University of Ulm
Ulm, Germany

Thomas Thüm
Paderborn University
Paderborn, Germany

ABSTRACT

Configurable systems often contain components from different fields or disciplines that are relevant for distinct stakeholders. For instance, tests or analyses targeting interactions of the software of a cyber-physical system may be only applicable for software components. However, managing such components in isolation is not trivial due, for instance, interdependencies between features. Feature models are a common formalism to specify such dependencies. Feature-model slicing corresponds to creating a subset of the feature model (e.g., with only components relevant to a particular stakeholder) that still preserves transitive dependencies from discarded features. However, slicing is computationally expensive and subsequent analyses often depend on complex computations, such as SAT or #SAT. With knowledge compilation, the original feature model can be translated to a beneficial format (e.g., d-DNNF or BDD) with an initial effort that accelerates subsequent analyses. Consequentially, acquiring a sliced target format depends on two expensive subsequent algorithms. In this work, we merge both steps by proposing *projected d-DNNF compilation*; a novel way to slice feature models that coincidently performs knowledge compilation to d-DNNF. Our empirical evaluation on real-world feature models shows that our tool pd4 often reduces runtimes substantially compared to existing techniques and scales to more input instances.

KEYWORDS

feature models, configurable systems, product lines, d-DNNF, knowledge compilation, slicing, projection

ACM Reference Format:
Chico Sundermann, Jacob Loth, and Thomas Thüm. 2024. Efficient Slicing of Feature Models via Projected d-DNNF Compilation. In *39th IEEE/ACM International Conference on Automated Software Engineering (ASE '24), October 27-November 1, 2024, Sacramento, CA, USA.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3691620.3695594

1 INTRODUCTION

Developing configurable systems demands various tests or analyses during different stages of a system's lifecycle [1, 50, 71]. Typically, not every configuration of features (i.e., decomposable units of the system) represents a desired product because of constraints such as dependencies between those features [7]. Hence, performing tests or analyses reliably requires configuration knowledge [7], separating valid from invalid configurations. Feature models are an expressive and commonly used [29, 47] formalism, which serves as a formal specification and an overview for stakeholders.

In many cases, one is interested in analyzing only a subset of a configurable system [2, 28, 29, 62–64, 75]. For instance, in a cyber-physical system, some analyses may only be reasonable for software-related features. Still, in practice there are often constraints over non-software features that transitively influence the valid configurations of the software. Neglecting those constraints may produce invalid configurations considering the original feature model and, consequentially, to inconsistent analysis results. Feature-model slicing corresponds to deriving a feature model with only features of interest but keeping transitive constraints [2, 39, 63]. For the example, slicing can be used to create a feature model with only software-related features that accepts the same software configurations as the overall feature model.

Current techniques for feature-model slicing face scalability issues for larger feature models [2, 39]. In particular, the currently most scalable solution seems to be slicing by applying logical resolution to remove features [39]. Here, features are removed one-by-one which is computationally expensive when removing larger parts of the feature model. However, common use cases for slicing, such as providing views [62], or feature-model interfaces [63], or representing solution space [29], often require slicing large parts of the feature model, which limits the scalability of existing approaches.

Many analyses on the configuration knowledge depend on, potentially numerous, complex computations [71], such as SAT [7, 38, 65] and #SAT [31, 41, 72]. For instance, homogeneity, a metric that indicates the similarity of configurations, requires a #SAT computation for every single feature [30, 71]. When facing multiple computations, knowledge compilation can be applied [20, 71]. Here, the feature model is translated to another format with beneficial properties (e.g., BDDs [13]) in an offline phase. Then, different analyses can be more efficiently performed on the target format in an online phase [20]. A format that gained popularity for analyzing configurable systems over the last few years is the determnistic decomposable negation normal form (d-DNNF) [12, 17, 41, 66, 73]. The format supports linear-time (w.r.t. d-DNNF size) computations for analyses based on SAT, #SAT, and generating configurations [20, 71]. Recent results show that d-DNNFs can vastly improve the scalability of such analyses on configurable systems [12, 66, 73].

However, compiling to d-DNNF is typically expensive and does not scale for some complex systems, such as the Linux kernel [71]. Since slicing and d-DNNF compilation both fail to scale to more complex systems in practice [39, 71], the applicability of analyses relying on these techniques is still limited.



This work is licensed under a Creative Commons Attribution International 4.0 License.
ASE '24, October 27-November 1, 2024, Sacramento, CA, USA
© 2024 Copyright held by the owner/author(s).
ACM ISBN 979-8-4007-1248-7/24/10
https://doi.org/10.1145/3691620.3695594

In this work, we propose to combine the concept of slicing and knowledge compilation by introducing *projected d-DNNF compilation*. The idea is to accelerate the slicing process and directly produce a more beneficial representation (i.e., a d-DNNF). In particular, the resulting d-DNNF enables linear-time (w.r.t. number of nodes in the d-DNNF) analyses based on counting [73], sampling [66], or validity [12] on the feature-model slice. We implemented a prototype by adapting the regular d-DNNF compiler `d4` [45] which appears to outperform other compilers on feature models [71].

We study the performance of projected d-DNNF compilation compared to the state of the art in a large-scale empirical evaluation. As subject systems, we consider real feature models and slices applied in practice from our industry partner, feature models from literature [37, 55], and instances from the Model Counting Competition [25]. We compare our approach to slicing with resolution [39] and projected model counters [26]. The latter base on similar techniques as projected d-DNNF compilation, but produce only the projected model count instead of a reusable d-DNNF.

Overall, we make the following three contributions:

- **Projected d-DNNF Compilation.** We propose and present the concept of projected d-DNNF compilation which adapts regular d-DNNF compilation by slicing during the process.
- **Optimizations.** We present various heuristics that we apply to increase efficiency of the compilation, as the scalability of reasoning engines, such as knowledge compilers, heavily depends on employed heuristics [44, 45, 69].
- **Prototype.** We implement the concept and the heuristics by adapting the state-of-the-art compiler `d4`.[1]
- **Empirical Evaluation.** We compare the performance of our projected d-DNNF compiler `pd4` to the state of the art in feature-model slicing and projected model counting.[2]

2 MOTIVATING EXAMPLE

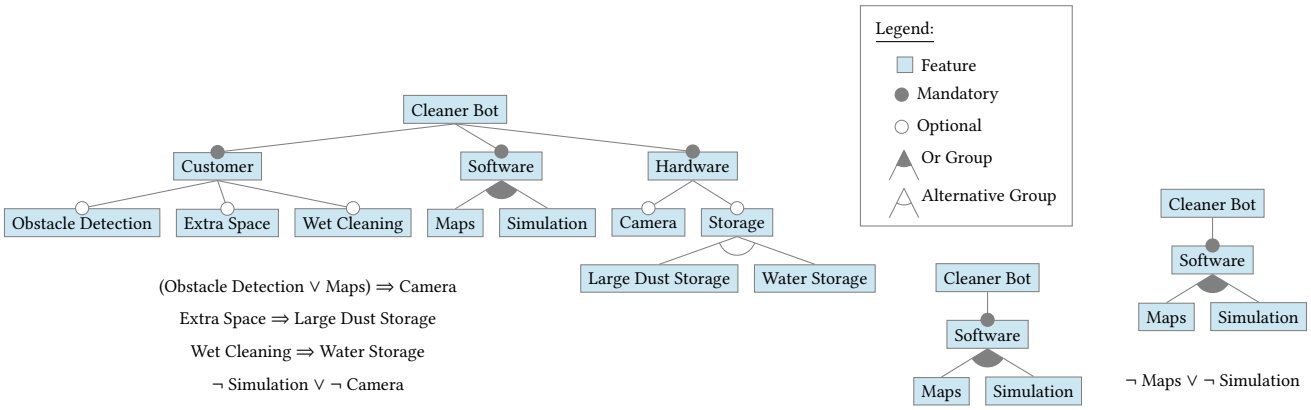
In the following, we provide a short example showcasing the benefits of slicing [2, 39]. Figure 1a depicts an exemplary feature model for a cleaning robot system. The configurable system consists of three major parts: `Customer`, `Software`, and `Hardware`. Each part must be configured as seen by their *mandatory* flag. A customer can select any combination of the three options `Obstacle Detection`, `Extra Space`, and `Wet Cleaning` as denoted by the *optional* flags. The *or*-relation below `Software` denotes that `Maps`, or `Simulation` or both can be selected. Further, a cleaner robot may have a `Camera` and exactly one `Storage` type as denoted by the *alternative*-relation. There are also several cross-tree constraints. For instance, `Wet Cleaning` requires having a `Water Storage`.

Typically, different stakeholders care only about specific parts of the configurable system. For instance, a software engineer may want to explicitly test variability only in source code without having to manage hardware or customer requirements. A typical use case for tests or analyses limited only parts of a system are software-in-the-loop tests [21], which examine software in a simulated environment without the designated hardware components. However, discarding the remaining features is not trivial. With the naive approach of just removing unwanted features and dependencies between them, transitive constraints can be overlooked. In our case, `Maps` and `Simulation` would be simultaneously selectable with the naive approach as seen in Figure 1b. In contrast, this combination would not be valid for the entire feature model due to interaction between `Camera`, `Simulation`, and `Maps` from the cross-tree constraints. ¬ `Simulation` ∨ ¬ `Camera` implies that only one of those can be selected, but `Maps` needs a `Camera` as indicated by the first cross-tree constraint. Consequentially, there are configurations that contain both `Maps` and `Simulation` that can be derived from this naive feature model, but are invalid for the complete feature model. This mismatch may induce problems when using or analyzing the naive model. For instance, the developer may build a test case for a software variant including both `Simulation` and `Camera`, which does not appear in any product of the system.

Slicing a feature model produces a sub feature model with transitive constraints as shown Figure 1c. Here, the effect of the interdependencies on the remaining features is specified as new constraint ¬ Maps ∨ ¬ Simulation. Similar problems also occur for the customer features. Only considering the `Customer` subtree, any combination of the three features can be selected. However, in practice, `Extra Space` and `Wet Cleaning` cannot be selected simultaneously due to storage types being alternative. While the described transitive effects could be recognized by humans for our small example, feature models in practice often have thousands of features and cross-tree constraints [8, 37, 41, 47]. Hence, automated tool support is needed to compute slices and also for subsequent analyses.

Various automated analyses have been proposed for feature models. As the output of slicing is also a feature model, these analyses are also applicable on the slice. Many of those analyses rely on solving complex computational problems [71], such as validity (SAT), counting (#SAT), and enumerating configuration (AllSAT). For instance, customers are likely only provided the `Customer` features for configuring their `Cleaner Bot`. Hence, using a model sliced on the customer features would be sensible for guiding the configuration process with decision propagation [40] or recommendations [49, 57]. A potential use case for counting on a sliced feature model is prioritizing features that appear in many products from the customers point of view [72]. Also, when testing software, it is common to create a sample of configurations [23, 38, 54] from a sliced software feature model [28], as the other features are not relevant for the software tests. The three listed computational problems, namely SAT, #SAT, and creating a sample of configurations, can be solved in linear time on a d-DNNF with respect to its number of nodes [20]. Furthermore, many automated analyses on configurable systems from various areas of software engineering rely on numerous of such complex computations [71]. For instance, providing information on selectable features during the configuration process with decision propagation requires $O(|\text{features}|)$ SAT invocations *per configuration step* [40]. Homogeneity [30, 72], a metric indicating the similarity of configurations, requires a #SAT invocation for every feature. Hence, having a projected d-DNNF can be beneficial to compute a feature-model slice and perform subsequent analyses on it.

[1] https://github.com/SoftVarE-Group/pd4
[2] https://doi.org/10.5281/zenodo.13752743



(a) Original **(b) Naive Submodel** **(c) Sliced Submodel**

Figure 1: Slicing a Cleaner Bot Product Line to Its Software Components

3 BACKGROUND

In this section, we introduce the following concepts that are relevant for this work: feature models, slicing, and d-DNNFs.

3.1 Feature Models

A *configurable system* is a family of related products that share certain *features*. Typically, not every *configuration* of features results in a desired product of the system. In this work, we use *feature models* as formalism to specify the set of valid configurations over those features [7, 36, 51]. The most common notation are *feature diagrams* which define features and dependencies between them using a tree hierarchy combined with cross-tree constraints. Our motivating example Figure 1a is visualized as feature diagram.

Formally, we use a much more generic specification of feature models without a concrete hierarchy. Rather, we define feature models $FM = (N, F)$ as tuple where N is the set of features and F a Boolean formula describing the set of valid configurations. Hence, our findings are applicable also other representations of configuration knowledge as long as they can be reduced to Boolean constraints over the features. We limit the expressiveness to Boolean logic here, since cross-tree constraints are commonly restricted to Boolean logic in the literature [5, 42, 51] and translations from the tree hierarchy of feature diagrams to Boolean logic are well-known [16, 51]. Furthermore, there have been efforts to translate higher-order logics to Boolean logic [53]. During this work, we consider each formula to be in *conjunctive normal form* (CNF) for simplicity. A CNF is a conjunction of clauses where each clause is a disjunction of literals. CNFs are commonly used as input for reasoning engines and translations from feature models to CNF are well-known [42]. A *configuration* $C = (I, E)$ with $I, E \subseteq N$ and $I \cap E = \emptyset$ is *valid* if and only if the formula $F \wedge \bigwedge_{i \in I} i \wedge \bigwedge_{e \in E} \neg e$ is satisfiable. With $\mathbb{V}_{FM}$, we denote the set of valid configurations described by FM.

3.2 Slicing

Feature-model slicing refers to removing features from a feature model while preserving transitive dependencies [2, 39]. In the logic domain, the same concept is also known as *projection* [26] and *forgetting* [20]. We consider every removed variable $n \in N \setminus N'$ as *sliced* and every kept variable $n \in N'$ as *projected* during the remainder of this work. Formally, we call $FM' = (N', F')$ a slice of a feature model $FM = (N, F)$ if $N' \subset N$, Equation 1, and Equation 2 hold. First, if we reduce a valid configuration for FM by removing features that are not part of N', it is valid for the slice FM'. Second, for every valid configuration of FM' we can find at least one valid configuration for FM by selecting or deselecting sliced features.

$$\forall (I, E) \in \mathbb{V}_{FM} : (I \cap N', E \cap N') \in \mathbb{V}_{FM'} \tag{1}$$

$$\forall (I', E') \in \mathbb{V}_{FM'}, \exists (I, E) \in \mathbb{V}_{FM} : I' \subseteq I, E' \subseteq E \tag{2}$$

3.3 Current Approaches for Slicing

In this section, we very shortly present existing approaches for feature-model slicing. For a more thorough comparison, we refer to the work targeting those approaches [2, 39, 75].

Existential Quantification. With existential quantification, a variable is sliced by duplicating the original formula and replacing the variable with true and false in the formulas, respectively [39, 75]. The approach bases on the equivalence $\phi \equiv (\phi \wedge v) \vee (\phi \wedge \bar{v})$. In the left subformula, every v can then be replaced with $\top$ and $\bar{v}$ with $\bot$ and vice versa for the right subformula, resulting in a sliced formula $\phi_v \vee \phi_{\bar{v}}$ without the variable v. Then, the inserted $\top$, and $\bot$ can be resolved with unit propagation. The procedure can be optimized by only duplicating clauses that contain the current variable v. Still, for a slice $FM' = (N', F')$ of $FM = (N, F)$, we need to apply existential quantification for every single sliced variable $v \in N \setminus N'$, which increases the required effort for every variable.

Logical Resolution. With *logical resolution*, we can slice variables v by *resolving* each pair of clauses where one contains a positive literal v and the other contains a negative literal $\neg v$ [39]. Equation 3 shows two clauses with v and $\neg v$ respectively which equivalently resolve to the clause below which does not include v. In the formula, we then remove the premise clauses and add the resolvent clauses.

We need to resolve $|C_v| * |C_{\bar{v}}|$ clause pairs for each variable to slice, where C_v are clauses with v and $C_{\bar{v}}$ clauses with $\bar{v}$. Hence, just as for existential quantification, the required effort increases with the number of sliced features.

$$\frac{(v \vee a_1 \vee \ldots \vee a_n) \quad (\neg v \vee b_1 \vee \ldots \vee b_m)}{(a_1 \vee \ldots \vee a_n \vee b_1 \vee \ldots \vee b_m)} \tag{3}$$

BDD Projection. *Binary decision diagrams* (BDDs) can be used to slice a variable as one can disjunct two BDDs and apply a conditioning (i.e., set variables to $\top$ or $\bot$) in polynomial size and time [2, 20]. Hence, you can apply existential quantification by building the BDD $\beta_{slice} = (\beta \wedge v) \vee (\beta \wedge \neg v)$ from the original BDD β. Following this procedure, the BDD needs to be duplicated and disjuncted for every sliced variable which increases the computational effort per variable. In particular, the BDD may grow exponentially in the worst case as it is duplicated for every variable.

3.4 d-DNNF

A *d-DNNF* is a negation normal form that is also *deterministic* and *decomposable* [18]. Deterministic means that for all disjunctions each pair of disjuncts shares no satisfying assignments (i.e., $\forall d_i, d_j, i \neq j : d_i \wedge d_j \equiv \bot$). A formula is decomposable if for every conjunction the conjuncts share no variables. The combination of these properties enables linear time algorithms on the d-DNNF (w.r.t. its size) for SAT, #SAT, and enumeration of satisfying assignments [20]. Various feature-model analyses depend on solving potentially many of those computational problems [12, 66, 71, 73]. Hence, the initial effort of *compiling* to d-DNNF can often be amortized and yield runtime benefits [12, 66, 73]. Often, *decision-DNNFs* are considered which are a *specialization* of d-DNNFs. Here, each disjunction is a *decision node* which has the form $d_v = (v \wedge \phi) \vee (\bar{v} \wedge \psi)$. In that example, v is the *decision variable* of the decision node. A decision node is by definition always deterministic. Since decision-DNNF are a special case of d-DNNFs, the same computational problems can be solved in linear time as for d-DNNFs.

d-DNNF Compilation. With current d-DNNF compilers [18, 45, 52], the d-DNNF is built by using the *trace* of an *exhaustive DPLL traversal*. All available compilers use a *conjunctive normal form* (CNF) [18, 45, 52] as input. Figure 2 shows the trace for a traversal on the CNF $(a \vee \bar{b}) \wedge (a \vee c) \wedge (\bar{a} \vee b \vee c \vee d)$. During the iterative branching process, remaining variables are assigned true and false in one of the branches, respectively, until every clause is satisfied. For instance, starting with the original formula assigning a (left branch in Figure 2) results in the formula $b \vee \bar{c} \vee d$ which is then used for further branching.

A common optimization is using *connected components*. A connected component is a subset S of the clauses C where no variable $v \in S$ is part of $C \setminus S$. Consequentially, each connected component does not influence the other connected components and the branching can be performed separately for each component. An example can be seen in Figure 2 with $\bar{b} \wedge c$. Here, we have two connected components $\{\bar{b}\}$ and $\{c\}$.

The trace depicted in Figure 2 can be used to construct a d-DNNF. The resulting d-DNNF is shown in Figure 3. Every time the algorithm branches on a variable v, a disjunction is created $(v \wedge \phi) \vee (\bar{v} \wedge \psi)$, which is essentially a decision node on v. As every solution in the left branch contains a and every solution on the right branch does not, the disjunction is deterministic by construction. When the conjunction is split into different connected components, we add a conjunction in our d-DNNF which is decomposable by construction. In the remainder of this work, we call a d-DNNF compilation that keeps all variables as *complete*, and a compilation that slices variables as *projected*.

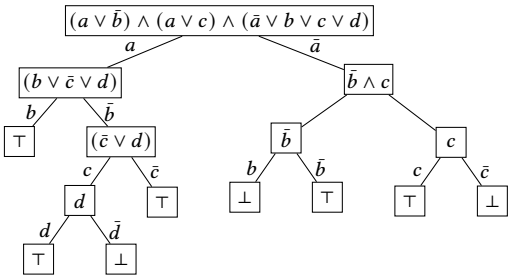


Figure 2: Exhaustive DPLL Trace

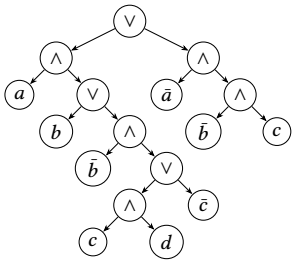


Figure 3: d-DNNF Constructed from Figure 2

4 PROJECTED D-DNNF COMPILATION

In this work, we introduce *projected d-DNNF compilation*, a novel way to acquire projected d-DNNFs in one step. Figure 4 shows the idea of our approach. Instead of subsequently using slicing and then d-DNNF compilation, our proposal combines them into one step, directly producing a projected d-DNNF. We realize this idea by adapting regular d-DNNF compilation to only consider projected variables. Instead of using the trace of a full exhaustive DPLL traversal as shown in Figure 2, we discard sliced variables during the traversal. In theory, our approach can be applied to every propositional formula. The d-DNNF properties (determinism and decomposability) are not restrictions on the input formula but rather guarantees by our procedure on the resulting knowledge compilation artifact. The only potential limitation for propositional instances is scalability as the underlying problem is computationally complex. In this section, we first explain the base algorithm and then discuss an intuition for its correctness.

4.1 Base Algorithm

Algorithm 1 shows our base algorithm to perform projected d-DNNF compilation which is an exhaustive DPLL traversal with the following adaptations. First, we only branch on projected variables (cf. line 7), as otherwise decisions on the sliced variables may have

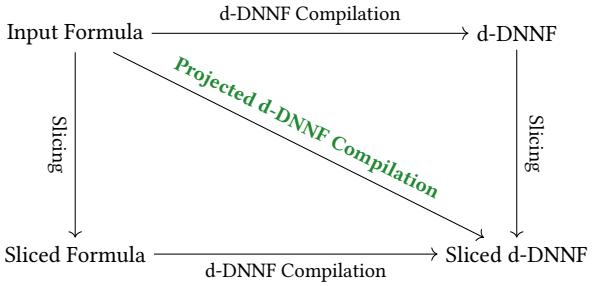


Figure 4: Commuting Diagram Showcasing Projected d-DNNF Compilation

Algorithm 1 Projected d-DNNF Compilation

Input: CNF F, set of projected variables pV
Output: projected d-DNNF

```
1:  procedure PD-DNNF(F, pV)
2:      if !SAT(F) then return ⊥
3:      subNodes ← ∅
4:      for φ in connectedComponents(F) do
5:          if vars(φ) ∩ pV ≠ ∅ then        ▷ Includes projected vars
6:              d_v ← nextDecisionVariable(φ) ▷ d_v ∈ vars(F) ∩ pV
7:              D_φ ← PD-DNNF(φ ∧ d_v) ∨ PD-DNNF(φ ∧ d̄_v)
8:              subNodes ← subNodes ∪ D_φ
9:      if |subNodes| = 0 then return ⊤      ▷ All variables sliced
10:     return ⋀_{D∈subNodes} D
```

transitive impact on the satisfying assignments of projected variables. Second, we ignore connected components that only consist of sliced variables (cf. line 5). The main procedure Algorithm 1 uses recursion to compute a partial d-DNNF for the current CNF. If the CNF is not satisfiable, we always return a $\bot$-node for the current CNF. Otherwise, we divide the CNF into the possible connected components (possibly only one) which are sets of clauses that share no variables with any other connected component (cf. Section 3.4). Then, we discard each component that includes only sliced variables. Note that each connected component is satisfiable, as we would otherwise directly return $\bot$ (cf. line 2).

For each component including at least one projected variable, we branch on one of the projected variables. We then recursively compute the partial d-DNNFs for both branches. A disjunction over the d-DNNFs resulting from both branches (cf. line 8) is semantically equivalent to the currently considered connected component as $\phi \equiv (\phi \wedge d_v) \vee (\phi \wedge d_{\bar{v}})$. A conjunction over these disjunctions is semantically equivalent to the input CNF F and is the result of the procedure `pd-DNNF`. Note that selecting promising variables for branching is a vital aspect for performance. We discuss our selection heuristics in detail in Section 5.

Figure 5 shows the impact of the adaptations on the DPLL trace of our running example. Here, we slice (i.e., remove) the variables c and d. In the first step, we have exactly one connected component and, thus, branch on one of the variables. Here, we just need to ensure that we branch on one of the projected variables. In the right

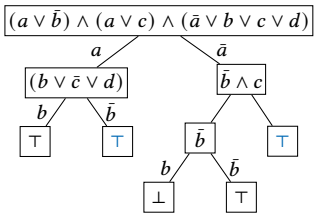


Figure 5: Projected Exhaustive DPLL Trace for Slicing c & d

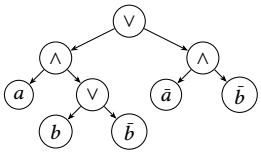


Figure 6: Projected d-DNNF Constructed from Figure 5

branch where we set $\bar{a}$, decision propagation leads to the formula $\bar{b} \wedge c$ which consists of two connected components. The right one (c) only contains sliced variables, and we replace it with $\top$ since it is satisfiable. Similarly, on the left side, after we branch on $\bar{b}$, we only have one connected component $\bar{c} \vee d$, which only contains sliced variables. Instead of further branching on $(\bar{c} \vee d)$ and c as done in Figure 2, we check its satisfiability and replace it with $\top$. We can apply the same rules as shown in Section 3.4 to convert the trace (i.e., Figure 5) to a d-DNNF. The resulting projected d-DNNF can be seen in Figure 6.

4.2 Correctness

In the following, we provide an intuition why Algorithm 1 computes a correctly projected d-DNNF. To this end, we assume that the regular d-DNNF compilation [45] shown in Section 3.4 is correct. By construction, just as regular compilation, our algorithm guarantees determinism for every disjunction and decomposability for every conjunction. It remains to show that the resulting projected d-DNNF D' is indeed a slice of the full d-DNNF D following the definition in Section 3.2. In particular, we discuss why (1) every satisfying assignment of D is satisfiable for D', when reduced to the variables in D' and (2) for every satisfying assignment of D', there is an extension with sliced variables that is satisfiable. There are two main differences between the respective base algorithms for regular and projected d-DNNF compilation: further restrictions on the variable ordering and the handling of connected components that only contain sliced variables. Since variable orderings used for projected compilation are always valid for a regular d-DNNF compilation, we assume that the same variable orderings are used for simplicity. Note that the algorithm should always produce semantically equivalent d-DNNFs for different variable orderings. Consider the current subformula $F = \phi_1 \wedge \ldots \wedge \phi_s \wedge \ldots \wedge \phi_z$ with ϕ_s being a connected component with only sliced variables. If $\text{SAT}(\phi_s) \equiv \top$, we have $D'_F \equiv \phi_1 \wedge \ldots \wedge \top \wedge \ldots \wedge \phi_z$ and $D_F \equiv \phi_1 \wedge \ldots \wedge \phi_s \wedge \ldots \wedge \phi_z$. It follows that all variables appearing in the connected components $\forall i \neq s : \phi_i$ can be equally assigned for D'_F and D_F. Hence, for an assignment satisfying for D_F, we can remove the variables from

ϕ_s to derive a satisfying assignment for D'_F, which fulfills (1). For every assignment in D'_F, we can use any satisfying assignment on the variables from ϕ_s to find a satisfying assignment for D_F, which fulfills (2). If SAT(ϕ_s) $\equiv \bot$, we have $D'_F \equiv \bot \equiv D_F$ and, thus, the same empty set of satisfying assignments which fulfills both conditions (1) and (2). Overall, it follows that our handling of connected components for slicing indeed produces a slice of the d-DNNF resulting from compilation without the adaptation. Since the correctness of the concrete implementation and employed optimizations is also relevant for usage in practice, we also study the correctness empirically in Section 6.

5 OPTIMIZATIONS

The performance of common reasoning engines, such as SAT [9] solvers and #SAT [14, 67, 76] solvers, and knowledge compilers [18, 19, 45, 52] heavily rely on heuristics. Even though solvers are based on the same base algorithms (e.g., DPLL or CDCL [48]), there are substantial differences regarding their scalability for feature models [58, 70]. In this section, we present the different heuristics we apply for optimizing the base approach presented in Section 4.

5.1 Preprocessings

As initial step, we perform different preprocessings that simplify the formula to accelerate the actual compilation. Preprocessings are a common strategy employed in reasoning engines [43–45, 69, 76]. Here, the formula is adapted so that the main algorithm still produces a semantically equivalent result but potentially with less resources, such as runtime or memory. For projected d-DNNF compilation, we collected various preprocessings from projected model counters [43, 67, 69]. However, several popular preprocessings are not trivially applicable for projected d-DNNF compilation as they only preserve the model count but not equivalence [42, 43]. Consequentially, subsequent analyses on the d-DNNF may produce faulty results. In the following, we discuss the preprocessings we apply and adaptations we made to prevent information loss.

Equivalent Literal Substitution. Two literals are equivalent if they evaluate to the same truth value ($\top$ or $\bot$) for every satisfying assignment [46]. For a set of equivalent literals $\{l_1, \ldots, l_n\}$, we can substitute each literal with a representative from the set. Note that the literals can be either positive or negative. For instance, we can replace each of $l_2, \ldots l_n$ with l_1 and each of $\bar{l_2}, \ldots \bar{l_n}$ with $\bar{l_1}$. In order to reason about all features on the compiled d-DNNF, we however need to keep track of substituted literals. We realize this by storing the equivalence sets and replacing the representative literal l_1 with $l_1 \wedge l_2 \wedge \ldots \wedge l_n$ and $\bar{l_1}$ with $\bar{l_1} \wedge \bar{l_2} \wedge \ldots \wedge \bar{l_n}$ in the compiled d-DNNF. Without those replacements, we would lose the information of the substituted equivalent literals and could not consider the corresponding variables in analyses on the d-DNNF. For example, we could not query how many satisfying assignments include l_2.

Partial Resolution. As described in Section 3.3, logical resolution can be used to slice variables. However, in the worst case we have a quadratic increase of clauses *per variable* to be sliced. The idea of *partial resolution*, which is used in the projected model counter GPMC,[3] is to resolve only sliced variables that do not increase the number of clauses or the *tree-width* [24] of the formula. Projected model counting has exponential time complexity w.r.t. tree-width, which is a metric that describes how close a graph is to a tree [26]. To measure the tree-width, the formula is represented as undirected graph where each variable is a node and there is an edge between those nodes if both variables appear together in a clause. After applying partial resolution, we use the simplified formula with some variables already sliced for the projected d-DNNF compilation.

Backbone Propagation. The *backbone* of a propositional formula F describes the literals that need to be satisfied in order for the formula to be satisfiable [34]. Formally, a literal l is part of the backbone if $\bar{l} \wedge F \equiv \bot$. Hence, the formula F can be simplified by replacing all l with $\top$ and $\bar{l}$ with $\bot$ and then perform decision propagation. With decision propagation, clauses that contain a $\top$ are removed as they are satisfied and $\bot$ is removed from clauses. Then, if a new unit clause was produced, the respective variable is again substituted with $\top$ and $\bot$ until a fixpoint is reached. Again, for projected d-DNNF compilation, we need to keep track of those variables to produce a complete d-DNNF. After the compilation, we attach the literals to the d-DNNF D that are part of the backbone B as $D \wedge \bigwedge_{l \in B} l$. Note that this top level conjunction is always decomposable as the variable backbones are not part of D as they were temporarily discarded during the compilation.

5.2 Weighted Hypergraph Partitioning

Connected components (i.e., sets of clauses that do not share variables) can be processed separately without worrying about interdependencies [6, 76]. Consequentially, a strategy often followed by #SAT solvers [6, 59, 76] and d-DNNF compilers [45] is finding *variable orderings* that split the formula to multiple connected components within only few decisions.

The best performing d-DNNF compiler for feature models [71] `d4` [45] uses *hypergraph partitioning* for deriving a variable order. Here, the CNF is represented as a hypergraph with each node being a clause. The hyperedges correspond to one variable each and indicate which clauses the variable occurs in. With hypergraph partitioning, we aim to find a *cut* which is a set of variables/edges that, when removed, split the hypergraph in two partitions with similar size. After deciding all variables from the cut in branching, our CNF will also be split into distinct connected components.

Regular hypergraph partitioning, such as used in `d4`, is not applicable for projected d-DNNF compilation. First, we cannot remove sliced variables for the partitioning as assigning them during branching may have a transitive impact on the projected variables. Second, we cannot ignore their dependencies as this could result in components that share sliced variables. Then, the components are not independent and, thus, cannot be processed separately.

To prevent the partitioning from using sliced variables while still considering their dependencies, we propose to use *weighted hypergraph partitioning* [61]. With that generalization of hypergraph partitioning, each edge is assigned a weight (or cost). The algorithm then tries to find a partition with a low cost of removed edges. We set the weight of projected variables to a low constant and the sliced variables to a very high constant. With high enough weights, the sliced variables are not selected but also not disregarded if they connect partitions.

[3] https://git.trs.css.i.nagoya-u.ac.jp/k-hasimt/GPMC

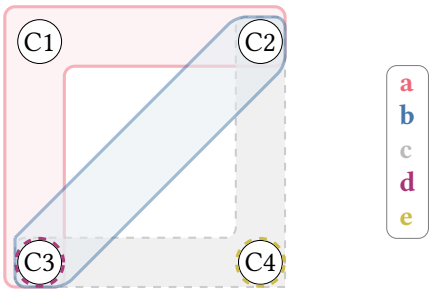


Figure 7: Weighted Hypergraph

Figure 7 shows an example for a weighted hypergraph for an adapted variant of our running example $F' = (a \vee \bar{b}) \wedge (a \vee c) \wedge (\bar{a} \vee b \vee c \vee d) \wedge (c \vee e)$. The four nodes C1-C4 represent the clauses of F'. The edges (or boxes) represent the variables. For instances, the red box represents the variable a, which is part of the clauses C1, C2, and C3. Projected variables are represented by solid lines (low weights) and sliced variables by dashed lines (high weights). In our example Figure 7, both a and c would partition the formula into two parts, respectively. However, due to the weighting the partitioning algorithm does not consider c as cutting variable. Removing a partitions the graph into two components {C1} and {C2, C3, C4} which then can be separately compiled to d-DNNF. Consequentially, our procedure would use a first in the variable ordering for branching. Note that it may be necessary to remove multiple variables before achieving a partitioning. Just as regular d4, we use hypergraph partitioning at the start of the compilation procedure and also dynamically on demand during the compilation.

As an optimization, we can use *equivalent variables* to also simplify the hypergraph. In the hypergraph, we replace the set of edges corresponding to equivalent variables with one edge. This comes with two main advantages: First, the hypergraph contains fewer edges which simplifies partitioning. Second, if the set of equivalent variables contains at least one projected variable, we can assign a low weight to this variable, as all sliced variables are implicitly decided by branching on the projected variable. Note that the initial hypergraph does not contain equivalent variables if we use our preprocessing to substitute equivalent literals. However, simpifying the hypergraph with equivalent variables is applicable for dynamic invocations of the partitioning or if the preprocessing is turned off.

5.3 Other Optimizations

Variable Ordering. With weighted hypergraph partitioning, we identify a set of variables to prioritize during branching. Still, the runtimes of the compiler depend on specific *variable orderings* within this set. For our compiler, we use a slightly adapted version of the commonly used [45, 67] *variable state aware decaying sum (VSADS)* [59]. VSADS combines two heuristics that prioritize (1) variables appearing in many clauses (i.e., DLCS) and (2) variables that satisfy recently identified conflict clauses (i.e., VSIDS) [59]. For our projected d-DNNF compilation, we adapt VSADS to only consider projected variables.

Pure Literal Elimination. A literal l is pure if $\bar{l}$ does not occur in the formula [35]. A pure literal can be trivially sliced during compilation as we can just assume $\top$ for positive or $\bot$ for negative literal as assignment. As no clause contains $\bar{l}$, the assumed assignment does not falsify any satisfying assignment over projected variables.

5.4 Tool Support

We realize the main concept and all the listed heuristics in our tool pd4, which is an extension of the popular d-DNNF compiler d4. We use d4 as recent work showed that d4 performs better on feature models than other available d-DNNF compilers [71]. pd4 takes CNFs in DIMACS format as input with an additional header for specifying the projected variables. The header follows the syntax `c p show v1 v2 v3 ...` with `v1`–`v3` being the variables to project. This header is also used in the projected model counting track of the Model Counting Competition [25]. pd4 is available at GitHub.[4]

6 EVALUATION

In our empirical evaluation, we examine the performance of our proposal pd4 compared to state-of-the-art tools. A replication package including solvers, data (excluding confidential models), and scripts to reproduce the experiments is publicly available.[5]

6.1 Research Questions

RQ1 How does the performance of projected d-DNNF compilation compare to slicing combined with regular compilation?

For **RQ1**, we aim to compare performance of the current state-of-the-art for acquiring a projected d-DNNF. As we are the first to introduce projected d-DNNF compilation, we compare our prototype to a two-step approach that subsequently performs slicing and then d-DNNF compilation. For the comparison of performance, we consider overall runtime and size of the produced d-DNNFs. As many subsequent analyses base on traversals of the d-DNNF [66, 73], the size is also relevant for runtime of feature-model analyses when using the compiled d-DNNF.

RQ2 How does the performance of projected d-DNNF compilation compare to projected model counting?

With **RQ2**, we compare the performance of projected d-DNNF compilation to closely related state-of-the-art tools, namely projected model counters. While projected model counters only compute the number of satisfying assignments instead of a reusable d-DNNF, they rely on the same base technique as our approach (i.e., exhaustive DPLL traversal).

RQ3 How do optimizations impact the performance of pD4?

For **RQ3**, we perform an ablation study to examine the impact of the central optimizations we employ for our approach. In particular, we analyze the impact of the preprocessings and weighted hypergraph partitioning presented in Section 5.

6.2 Experiment Design

Subject Systems. We use three datasets shown in Table 1 for comparison to increase external validity. First, we use twelve real feature models with four slices each that are actually applied in

[4] https://github.com/SoftVarE-Group/pd4
[5] https://doi.org/10.5281/zenodo.13752743

Table 1: Subject Systems

Track	Instances	Variables	Clauses
Industrial Models & Slices	12 · 4	376–3,286	1,015–51,450
Literature Models & Random Slices	49 · 100	17–62,482	16–350,221
Competition Instances	200	100–3,423,788	545–6,163,392

Table 2: Evaluated Tools

Name	Type	Origin
pd4	Projected d-DNNF Compiler	This work
Slicing FeatureIDE (v3.10)	Slicer via Resolution	[39]
d4 (v2)	d-DNNF Compiler	[45]
GPMC (v1.1.1)	Projected Model Counter	[25]
d4-pmc (v2)	Projected Model Counter	[45]
arjun (MCC Version)	Projected Model Counter	[67, 69]

practice from our industry partner. The feature models represent configurable systems from the automotive domain and each slice represents a part of the feature model for different disciplines. Second, we consider publicly available real-world feature models from the literature and generate random sets of features to slice from them. Here, we generate 100 slices for every feature model with each slice removing between 10% and 80% of the variables. Third, we compare the performances on instances of the Projected Model Counting Track from the Model Counting Competition 2022.[6]

Evaluated Tools. Table 2 shows the tools we consider for comparison in our experiments. First, we evaluate our projected d-DNNF compiler pd4. Second, we consider a combination of slicing as performed by FeatureIDE [39] and the regular d4 [45] compiler as baseline for acquiring a projected d-DNNF. Third, we compare the performance to the best performing projected model counters from the Projected Model Counting Track from the Model Counting Competition (MCC) 2022 and 2023, namely GPMC,[7] d4-pmc,[8] and arjun [69]. arjun is built on top of the solver sharpSAT [76].

Experiment Execution. For the experiments, we preprocessed every input instance as CNF in DIMACS format [10] with additional meta information indicating the variables that are part of the projection. We then execute each tool listed in Table 2 on every CNF projection representing a feature model. For each projection, we set a timeout of 10 minutes as (1) there were no substantial changes to the results by increasing the timeout in preliminary experiments and (2) the high number of instances are already computationally expensive with 10 minutes of timeout. For the models from the model counting competition, we use the same timeout as in the competition which is 60 minutes per instance. We repeat each measurement three times to reduce the effects of computational bias. If not stated otherwise, we present the median of those three repetitions in this section. For the combination of slicing and regular d4, we execute slicing first and then invoke d4 on the resulting sliced CNF. To reduce the demand of computational resources, we perform the ablation study on the different optimization variants only on the feature models where we have actual slices from industry available, because these are assumed to be more realistic for the research focus (i.e., feature-model slicing). To examine the correctness of tools, we compared resulting model counts. For the d-DNNFs approaches, we used ddnnife [73] to compute model counts on the compiled d-DNNFs.

[6] https://mccompetition.org/past_iterations
[7] https://git.trs.css.i.nagoya-u.ac.jp/k-hasimt/GPMC
[8] https://github.com/crillab/d4v2

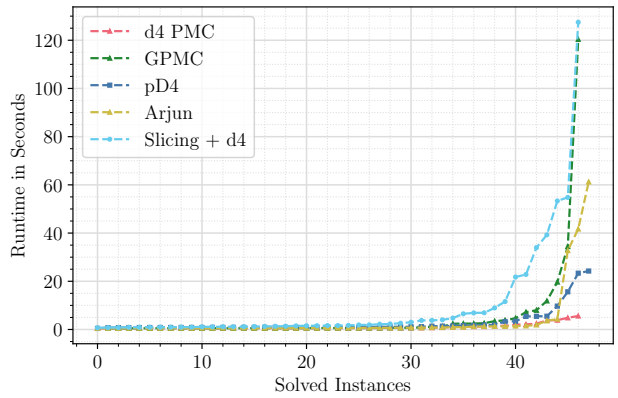
6.3 Results

Industry Models & Slices. Figure 8a shows a cactus plot indicating runtime of the evaluated tools on the industry models with real slices. The x-axis shows the number of solved instances and y-axis the runtime in seconds. Note that for each tool the runtimes are sorted individually on the x-axis. In general, the further to the right a line for a tool is drawn, the more instances have been solved by this tool. arjun and our projected d-DNNF compiler pd4 are able to solve to the most instances (47). For the 47 instances, pd4 requires 136 seconds and Arjun 176 seconds. Projected model counting with d4 (d4-pmc) only solves 46 instances but for the instances with respective lowest runtimes, d4-pmc is faster with 52.0 seconds while pd4 requires 112 seconds. The combination of slicing with FeatureIDE and d-DNNF compilation scales to 46 models as well but requires 454 seconds. For instances solved by both pd4 and slicing, pd4 is 4.05 overall times faster. All d-DNNFs compiled by pd4 during our experiments produced the same model counts as all projected model counters.

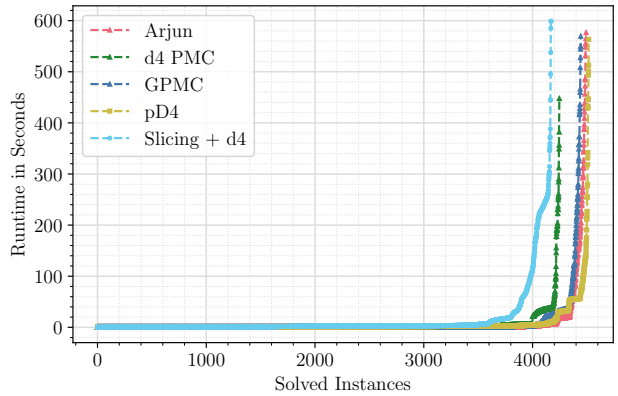
Literature Models & Random Slices. Figure 8b shows a cactus plot for the runtimes on the feature models from the literature and artificial slices. Our tool pd4 solved the most instances (4,516) followed by Arjun (4,492). While pd4 required 8.19 hours for the 4,516 instances, Arjun needed 8.86 hours for 4,492 instances. The combination of slicing and d4 successfully terminated for the fewest models (4,170). Note that the literature dataset contains various feature models with few features and constraints. Hence, every considered tool, even slicing, scales for most instances. In particular, every solver required less than one second each for the 3,000 instances inducing the respective shortest runtimes. For instances solved by both pd4 and slicing, pd4 is overall 9.25 times faster. We are also able to compile slices for feature models, for which regular d-DNNF compilation does not scale with available compilers [71]. For instance, pd4 successfully produced sliced d-DNNF for 96 of 100 slices for Linux (v2.6.33.3), but compiling to complete d-DNNF does not scale [71].

Competition Instances. Figure 8c shows a cactus plot for the runtimes on formulas of the Projected Model Counting track of the Model Counting Competition 2022. Again, our tool pd4 solves the highest number of instances (159) followed by GPMC (155). pd4 solves those 159 instances in 23 hours while GPMC requires 26 hours for 155. The combination of slicing and d4 successfully terminates for the fewest instances (52). For instances solved by both pd4 and slicing, pd4 is overall 22.0 times faster.

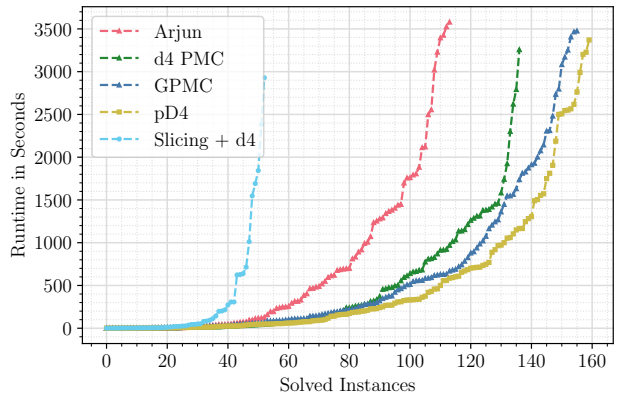
d-DNNF Sizes. Figure 9 shows the number of nodes for the d-DNNFs produced by pd4 and slicing combined with regular d4. The plots only include instances solved by both solvers. A marker below the blue line indicates that pd4 computed a smaller d-DNNF.



(a) Evaluated Tools on Industry Models with Real Slices



(b) Evaluated Tools on Models from Literature with Generated Slices



(c) Evaluated Tools on Instances from the MC2022

Figure 8: Runtimes of Evaluated Tools on Different Datasets

Overall, the sizes of the produced d-DNNFs are comparable. For feature models, pd4 tends to produce smaller d-DNNFs (for 61.7% of the instances). On the literature models, pd4 produces on average 5.07 times smaller d-DNNFs. On the industrial models, pd4 produces on average 25.1% smaller d-DNNFs. For the competition instances, slicing was able to compute considerably fewer d-DNNFs (53 vs 160), but slicing produces a smaller d-DNNF than pd4 in 78.8% of the cases for those successfully evaluated by both.

pd4 Optimizations. Figure 10 shows the impact of the different optimizations as a cactus plot for the industry instances. Turning off hypergraph partitioning has a smaller impact on the performance than preprocessing, but is still considerable. While the same number of instances are solved with and without partitioning, the overall runtimes for the solved instances are 136 seconds and 734 seconds (81.5% decrease), respectively. Without preprocessing, pd4 only solves 35 of the instances independently of using partitioning.

6.4 Discussion

RQ1 *How does the performance of projected d-DNNF compilation compare to slicing combined with regular compilation?* On all three datasets, the projected d-DNNF compilation with our tool pd4 substantially outperformed the subsequent execution of slicing and regular compilation. pd4 scales for far more instances than slicing followed by d-DNNF compilation. As slicing followed by invoking a regular d-DNNF compiler appears to be the only alternative that produces a projected d-DNNF, our results suggest that pd4 is the most promising approach to derive projected d-DNNFs. A further potential benefit is that we were able to compile projected d-DNNFs for which we could neither compile a complete d-DNNF nor perform slicing. The sizes of compiled d-DNNFs were comparable for both slicing with regular d4 and pd4, but pd4 generally produces smaller d-DNNFs for the feature-model instances.

RQ2 *How does the performance of projected d-DNNF compilation compare to projected model counting?* During all experiments, pd4 outperforms all projected model counters even though pd4 produces a d-DNNF for further reuse (e.g., [12, 66, 73]) instead of just the model count (i.e., a single number). Over the three datasets, pd4 scales for more instances than any projected model counter. For instances solved by all tools, pd4 is typically competitive with the projected model counters, but still sometimes slower. The largest observed difference is on the feature models from the literature where pd4 is 3.92 times slower than `Arjun` for the easier instances. Even for those cases, pd4 has considerable benefits as it returns both the model count and the d-DNNF which can be reused. The runtime advantages of pd4 over slicing and projected model counters on instances from the model counting competition indicate that pd4 is also promising for application domains beyond product lines and even if only the model count is needed.

RQ3 *How do optimizations impact the performance of pD4?* The performed ablation study strongly indicates that the employed optimizations positively impact the runtime of pd4. Weighted hypergraph partitioning substantially reduces runtimes, but pd4 still scaled for the same models without it. The preprocessings are crucial for the scalability, as pd4 scaled to considerably fewer instances without them.

6.5 Threats to Validity

External Validity of Slices. For the feature models from the literature, we have no information available on practically relevant slices. The randomly generated slices may not be representative



Figure 9: Number of Nodes for Produced d-DNNFs

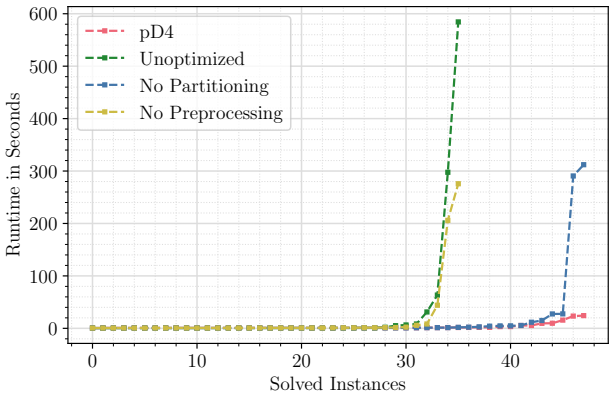


Figure 10: Impact of pD4 Optimizations

to slices needed in practice. The main issue is the availability of real-world slices on feature models. For the models from literature, we aim to mitigate the issue by acquiring a reasonable coverage with many slices of various sizes. Furthermore, we compare each tool on slices from practice provided by our industry partner and available projections from the model counting competition.

External Validity of Feature Models. Our selection of input instances may not be transferrable for all practically relevant feature models. To mitigate the issue, we collected feature models from various publications, domains, and with wide ranges of structural properties (e.g., 17–62,482 variables and 16–350,221 clauses). Furthermore, we examined the performance of our tool on instances beyond feature models, where `pd4` also performed well.

External Validity of Solver Parameterizations. Several tools used in our experiments offer parameters to control performance-relevant behavior of the algorithms. During our experiments, we invoked each solver with default parameters. Using other parameters may influence the performance of the different tools. However, examining different parameters vastly increases the complexity of the already computationally demanding evaluation due to combinatorial explosion of the parameters. Furthermore, we expect default parameters are a decent indicator for usage of these tools in practice.

Internal Validity. The measured runtimes of tools may differ between individual runs for the same input instance due to *computational bias*. To the best of our knowledge, each tool behaves deterministicly which reduces the impact of *random effects*. For more accurracy on the measured runtimes, we performed three repetitions per measurement. Furthermore, we argue that the multitude of input instances (5,148 per tool) also compensates for potential bias due to individual runs.

7 RELATED WORK

Projected Model Counting. Projected model counters compute the number of satisfying assignments induced by a subset of variables [26]. With our procedure, we essentially compute a projected d-DNNF from the trace of an exhaustive DPLL traversal which are also used by the projected model counters. Hence, the techniques are closely related. However, to the best of our knowledge, none of the available projected model counters supports producing a d-DNNF as output. Furthermore, many projected model counters use preprocessings that preserve the model count but not equivalence [67, 69]. Hence, the traces from the performed DPLL traversals cannot be directly used for a correct and complete d-DNNF. While we adapted some preprocessings to work for projected d-DNNF compilation (cf. Section 5), others, such as B+E [43] would require substantial computational effort to preserve equivalence.

d-DNNF Compilation. Three d-DNNF compilers are typically considered in the literature, namely `d4` [45], `dSharp` [52], and `c2d` [18]. The latest version of `d4`[9] supports projected model counting. However, to the best of our knowledge, no d-DNNF compiler supports compilation to a projected d-DNNF. In general, projected knowledge compilation has not been considered before in any domain. All listed compilers should be adaptable with our approach, we extended `d4` as it performed best for feature models in recent work [71].

[9] https://github.com/crillab/d4v2

d-DNNF Projection. It is possible to perform a projection on a given d-DNNF within linear runtime with respect to the number of d-DNNF nodes [3]. However, during the projection process the determinism is not preserved. The resulting format DNNF supports fewer linear-time queries for analyses [20]. For instance, counting in linear time complexity, which is one of the major selling points of d-DNNFs [20, 66, 73], is not possible a on DNNF. Aziz et al. [3] propose to (1) compile to d-DNNF with regular compilation, (2) perform a projection on that d-DNNF, (3) translate the resulting DNNF to CNF, and (4) compile this CNF to d-DNNF again. They also realized this in a prototype `d2c`, which we excluded for our experiments since it produced faulty results for the majority of instances in preliminary experiments. Also, this procedure requires to compute a d-DNNF over the entire feature model first, which is often expensive or may not even scale to complex models (cf., Figure 8b), such as the Linux kernel [71].

Feature-Model Slicing. Acher et al. [2] proposed slicing on binary decision diagrams. Slicing one variable can be performed in linear time and space with respect to the size of the BDD [20]. However, this procedure requires compiling to BDD first, which has been found to not scale for many feature models in recent work [32, 71]. Also, BDD compilation scales substantially worse than d-DNNF compilation for feature models [71]. Thüm et al. [75] use existential quantification to eliminate abstract (i.e., not relevant for the implementation) features from configuration to derive program variants. Krieter et al. [39] compare different slicing algorithms based on logical resolution. They integrated the best performing variant in FeatureIDE, which is also the variant we used in our empirical evaluation which showed that projected d-DNNF compilation scales to substantially more instances.

Feature-Model Analysis on d-DNNFs. The popularity of using d-DNNFs for feature-model analysis increased in the last ten years [12, 66, 73]. In recent work, we reduce a variety of feature-model analyses to d-DNNF counting operations. Sharma et al. [66] use d-DNNFs to compute uniform random samples. Bourhis et al. [12] compare using different one-time computation tools to d-DNNFs for counting, enumerating, and optimal configurations. The listed work operate on an existing d-DNNF while we propose the first algorithm to compile to projected d-DNNF. The plethora of analyses on d-DNNFs further motivates our work, as the proposed algorithms can also be applied to the projected d-DNNFs.

Projected Reasoning Beyond Product Lines. There are various relevant analyses in different application domains based on automated reasoning targeting projections [69]. Such analyses are mostly reduced to satisfiability, counting, and sampling (i.e., enumeration) [69], which all can be performed in polynomial runtime complexity on d-DNNFs. Duenas-Osorio et al. [22] use projected model counting for evaluating *network reliability*. Zawadzki et al. [77] employ projected model counting for *planning problems*. Teuber and Weigl [74] evaluate *software reliability* using projected model counting. In *computational biology*, Sashittal and El-Kebir [60] employ enumeration and counting on projected formulas representing transmission histories of disease outbreaks. Baluta et al. [4] verify different security-relevant properties of neural networks using repetitive invocations of projected model counting. Since the listed applications rely on operations that can be quickly performed on d-DNNFs, our procedure is potentially beneficial there, especially considering the strong performance on non-product-line instances in our evaluation.

d-DNNF Applications Beyond Product Lines. Various computationally complex problems can be solved on d-DNNFs in polynomial time with respect to their size [20]. Checking satisfiability and model counting, which are both relevant problems in many application domains [25, 27], can be computed even in linear time complexity. Hence, many application domains can benefit from the application of d-DNNFs. In addition to product lines [12, 66, 73], d-DNNFs have been applied in conformant planning [56], recommender systems [11], explainable machine learning [33], sequential diagnosis [68], and inference from bayesian networks [15]. In these domains, which already use d-DNNFs, our proposal may also be useful in the future if applications on projections emerge.

8 CONCLUSION

Typically, only parts of a configurable system are relevant for particular stakeholders. Slicing enables considering only a subset of features while preserving transitive dependencies. With our proposal of projected d-DNNF compilation, we tackle two issues of existing slicing approaches: the scalability of the slicing process and the runtime of subsequent analyses on the sliced feature model.

Our empirical evaluation strongly indicates that our prototype outperforms the state of the art of slicing and also even closely related projected model counters. Overall, projected d-DNNF compilation and our prototype `pd4` seem to be a promising solution for slicing, when subsequent analyses are required. Projected d-DNNF compilation may also be beneficial for other domains as our prototype also scales to more instances beyond feature models from the model counting competition. Since we found feature models for which we cannot compile a complete d-DNNF but a sliced one, this may enable additional analyses for those complex models.

ACKNOWLEDGMENTS

This work is based on the master's thesis of Loth. We thank Paul Bittner, Rahel Sundermann, and Tobias Heß for their valuable feedback on our work.

REFERENCES

[1] Iago Abal, Jean Melo, Stefan Stănciulescu, Claus Brabrand, Márcio Ribeiro, and Andrzej Wąsowski. 2018. Variability Bugs in Highly Configurable Systems: A Qualitative Analysis. *Trans. on Software Engineering and Methodology (TOSEM)* 26, 3, Article 10 (2018), 10:1–10:34 pages.

[2] Mathieu Acher, Philippe Collet, Philippe Lahire, and Robert B. France. 2011. Slicing Feature Models. In *Proc. Int'l Conf. on Automated Software Engineering (ASE)*. IEEE, 424–427.

[3] Rehan Abdul Aziz, Geoffrey Chu, Christian Muise, and Peter Stuckey. 2015. #∃ SAT: Projected Model Counting. In *Proc. Int'l Conf. on Theory and Applications of Satisfiability Testing (SAT)*. Springer, 121–137.

[4] Teodora Baluta, Shiqi Shen, Shweta Shinde, Kuldeep S. Meel, and Prateek Saxena. 2019. Quantitative Verification of Neural Networks and Its Security Applications. In *Proc. ACM SIGSAC Conf. on Computer and Communications Security (CCS)*. ACM, 1249–1264.

[5] Don Batory. 2005. Feature Models, Grammars, and Propositional Formulas. In *Proc. Int'l Systems and Software Product Line Conf. (SPLC)*. Springer, 7–20.

[6] Roberto J. Bayardo Jr. and Joseph Daniel Pehoushek. 2000. Counting Models Using Connected Components. In *Proc. Conf. on Artificial Intelligence (AAAI)*. AAAI Press, 157–162.

[7] David Benavides, Sergio Segura, and Antonio Ruiz-Cortés. 2010. Automated Analysis of Feature Models 20 Years Later: A Literature Review. *Information Systems* 35, 6 (2010), 615–708.

[8] Thorsten Berger, Steven She, Rafael Lotufo, Andrzej Wąsowski, and Krzysztof Czarnecki. 2013. A Study of Variability Models and Languages in the Systems Software Domain. *IEEE Trans. on Software Engineering (TSE)* 39, 12 (2013), 1611–1640.

[9] Armin Biere. 2008. PicoSAT Essentials. *J. Satisfiability, Boolean Modeling and Computation* 4 (2008), 75–97.

[10] Armin Biere, Marijn Heule, Hans van Maaren, and Tony Walsh. 2009. *Handbook of Satisfiability*. IOS Press.

[11] Pierre Bourhis, Laurence Duchien, Jérémie Dusart, Emmanuel Lonca, Pierre Marquis, and Clément Quinton. 2022. *Pseudo Polynomial-Time Top-k Algorithms for d-DNNF Circuits*. Technical Report. Cornell University Library.

[12] Pierre Bourhis, Laurence Duchien, Jérémie Dusart, Emmanuel Lonca, Pierre Marquis, and Clément Quinton. 2023. *Reasoning on Feature Models: Compilation-Based vs. Direct Approaches*. Technical Report. Cornell University Library.

[13] Randal E. Bryant. 2018. Binary Decision Diagrams. In *Handbook of Model Checking*. Springer, 191–217.

[14] Jan Burchard, Tobias Schubert, and Bernd Becker. 2015. Laissez-Faire Caching for Parallel #SAT Solving. In *Proc. Int'l Conf. on Theory and Applications of Satisfiability Testing (SAT)*. Springer, 46–61.

[15] Mark Chavira, Adnan Darwiche, and Manfred Jaeger. 2006. Compiling relational Bayesian networks for exact inference. *International Journal of Approximate Reasoning (IJAR)* 42 (2006), 4–20.

[16] Krzysztof Czarnecki and Andrzej Wąsowski. 2007. Feature Diagrams and Logics: There and Back Again. In *Proc. Int'l Systems and Software Product Line Conf. (SPLC)*. IEEE, 23–34.

[17] Adnan Darwiche. 2001. Decomposable Negation Normal Form. *J. ACM* 48, 4 (2001), 608–647.

[18] Adnan Darwiche. 2002. A Compiler for Deterministic, Decomposable Negation Normal Form. In *Proc. Conf. on Artificial Intelligence (AAAI)*. AAAI Press, 627–634.

[19] Adnan Darwiche. 2011. SDD: A New Canonical Representation of Propositional Knowledge Bases. In *International Joint Conferences on Artificial Intelligence*. AAAI Press, 819–826.

[20] Adnan Darwiche and Pierre Marquis. 2002. A Knowledge Compilation Map. *J. Artificial Intelligence Research (JAIR)* 17, 1 (2002), 229–264.

[21] Stephanie Demers, Praveen Gopalakrishnan, and Latha Kant. 2007. A Generic Solution to Software-in-the-Loop. In *IEEE Military Communications Conference (MILCOM)*. 1–6.

[22] Leonardo Duenas-Osorio, Kuldeep Meel, Roger Paredes, and Moshe Vardi. 2017. Counting-Based Reliability Estimation for Power-Transmission Grids. *Proc. Conf. on Artificial Intelligence (AAAI)* 31 (2017). https://doi.org/10.1609/aaai.v31i1.11178

[23] Rafael Dutra, Kevin Laeufer, Jonathan Bachrach, and Koushik Sen. 2018. Efficient Sampling of SAT Solutions for Testing. In *Proc. Int'l Conf. on Software Engineering (ICSE)*. ACM, 549–559.

[24] Aurélie Favier, Philippe Jégou, and Simon De Givry. 2011. Solution Counting for CSP and SAT With Large Tree-Width. *Control Systems and Computers (CSC)* 2 (2011), 4–13.

[25] Johannes K. Fichte, Markus Hecher, and Florim Hamiti. 2021. The Model Counting Competition 2020. *ACM J. of Experimental Algorithmics (JEA)* 26, Article 13 (2021), 26 pages.

[26] Johannes K. Fichte, Markus Hecher, Michael Morak, and Stefan Woltran. 2018. Exploiting Treewidth for Projected Model Counting and Its Limits. In *Proc. Int'l Conf. on Theory and Applications of Satisfiability Testing (SAT)*. Springer, Cham, Switzerland, 165–184.

[27] Jun Gu, Paul W Purdom, John Franco, and Benjamin W Wah. 1996. *Algorithms for the Satisfiability (SAT) Problem: A Survey*. Technical Report. Cincinnati University.

[28] Marc Hentze, Tobias Pett, Chico Sundermann, Sebastian Krieter, Thomas Thüm, and Ina Schaefer. 2022. Generic Solution-Space Sampling for Multi-Domain Product Lines. In *Proc. Int'l Conf. on Generative Programming: Concepts & Experiences (GPCE)*. ACM.

[29] Marc Hentze, Chico Sundermann, Thomas Thüm, and Ina Schaefer. 2022. Quantifying the Variability Mismatch Between Problem and Solution Space. In *Proc. Int'l Conf. on Model Driven Engineering Languages and Systems (MODELS)*. IEEE, 322–333.

[30] Ruben Heradio, David Fernández-Amorós, Christoph Mayr-Dorn, and Alexander Egyed. 2019. Supporting the Statistical Analysis of Variability Models. In *Proc. Int'l Conf. on Software Engineering (ICSE)*. IEEE, 843–853.

[31] Rubén Heradio-Gil, David Fernández-Amorós, José Antonio Cerrada, and Carlos Cerrada. 2011. Supporting Commonality-Based Analysis of Software Product Lines. *IET Software* 5, 6 (2011), 496–509.

[32] Tobias Heß, Chico Sundermann, and Thomas Thüm. 2021. On the Scalability of Building Binary Decision Diagrams for Current Feature Models. In *Proc. Int'l Systems and Software Product Line Conf. (SPLC)*. ACM, 131–135.

[33] Xuanxiang Huang, Yacine Izza, Alexey Ignatiev, Martin Cooper, Nicholas Asher, and Joao Marques-Silva. 2022. Tractable Explanations for d-DNNF Classifiers. In *Proc. Conf. on Artificial Intelligence (AAAI)*, Vol. 36. AAAI Press, 5719–5728.

[34] Mikoláš Janota, Inês Lynce, and Joao Marques-Silva. 2015. Algorithms for Computing Backbones of Propositional Formulae. *AI Communications* 28, 2 (2015), 161–177.

[35] Jan Johannsen. 2005. The Complexity of Pure Literal Elimination. *J. Autom. Reason.* 35 (2005), 89–95.

[36] Kyo C. Kang, Sholom G. Cohen, James A. Hess, William E. Novak, and A. Spencer Peterson. 1990. *Feature-Oriented Domain Analysis (FODA) Feasibility Study*. Technical Report CMU/SEI-90-TR-21. Software Engineering Institute.

[37] Alexander Knüppel, Thomas Thüm, Stephan Mennicke, Jens Meinicke, and Ina Schaefer. 2017. Is There a Mismatch Between Real-World Feature Models and Product-Line Research?. In *Proc. Europ. Software Engineering Conf./Foundations of Software Engineering (ESEC/FSE)*. ACM, 291–302.

[38] Sebastian Krieter. 2020. Large-Scale T-Wise Interaction Sampling Using YASA. In *Proc. Int'l Systems and Software Product Line Conf. (SPLC)*. ACM, 29:1–29:4.

[39] Sebastian Krieter, Reimar Schröter, Thomas Thüm, Wolfram Fenske, and Gunter Saake. 2016. Comparing Algorithms for Efficient Feature-Model Slicing. In *Proc. Int'l Systems and Software Product Line Conf. (SPLC)*. ACM, 60–64.

[40] Sebastian Krieter, Thomas Thüm, Sandro Schulze, Reimar Schröter, and Gunter Saake. 2018. Propagating Configuration Decisions With Modal Implication Graphs. In *Proc. Int'l Conf. on Software Engineering (ICSE)*. ACM, 898–909.

[41] Andreas Kübler, Christoph Zengler, and Wolfgang Küchlin. 2010. Model Counting in Product Configuration. In *Proc. Int'l Workshop on Logics for Component Configuration (LoCoCo)*. Open Publishing Association, 44–53.

[42] Elias Kuiter, Sebastian Krieter, Chico Sundermann, Thomas Thüm, and Gunter Saake. 2022. Tseitin or not Tseitin? The Impact of CNF Transformations on Feature-Model Analyses. In *Proc. Int'l Conf. on Automated Software Engineering (ASE)*. ACM, 110:1–110:13.

[43] Jean-Marie Lagniez, Emmanue Lonca, and Pierre Marquis. 2016. Improving Model Counting by Leveraging Definability. In *Proc. Int'l Joint Conf. on Artificial Intelligence (IJCAI)*. AAAI Press, 751–757.

[44] Jean-Marie Lagniez and Pierre Marquis. 2014. Preprocessing for Propositional Model Counting. *Proc. Conf. on Artificial Intelligence (AAAI)* 28, 1 (2014).

[45] Jean-Marie Lagniez and Pierre Marquis. 2017. An Improved Decision-DNNF Compiler. In *Proc. Int'l Joint Conf. on Artificial Intelligence (IJCAI)*. International Joint Conferences on Artificial Intelligence, 667–673.

[46] Yong Lai, Kuldeep S Meel, and Roland HC Yap. 2021. The Power of Literal Equivalence in Model Counting. In *Proc. Conf. on Artificial Intelligence (AAAI)*, Vol. 35. AAAI Press, 3851–3859.

[47] Rafael Lotufo, Steven She, Thorsten Berger, Krzysztof Czarnecki, and Andrzej Wąsowski. 2010. Evolution of the Linux Kernel Variability Model. In *Proc. Int'l Systems and Software Product Line Conf. (SPLC)*. Springer, 136–150.

[48] Joao Marques-Silva, Inês Lynce, and Sharad Malik. 2009. Conflict-Driven Clause Learning SAT Solvers. In *Handbook of Satisfiability*. IOS Press, 131–153.

[49] Raúl Mazo, Cosmin Dumitrescu, Camille Salinesi, and Daniel Diaz. 2014. Recommendation Heuristics for Improving Product Line Configuration Processes. In *Recommendation Systems in Software Engineering*. Springer, 511–537.

[50] Flávio Medeiros, Christian Kästner, Márcio Ribeiro, Rohit Gheyi, and Sven Apel. 2016. A Comparison of 10 Sampling Algorithms for Configurable Systems. In *Proc. Int'l Conf. on Software Engineering (ICSE)*. ACM, 643–654.

[51] Marcílio Mendonça, Andrzej Wąsowski, and Krzysztof Czarnecki. 2009. SAT-Based Analysis of Feature Models Is Easy. In *Proc. Int'l Systems and Software Product Line Conf. (SPLC)*. Software Engineering Institute, 231–240.

[52] Christian Muise, Sheila A. McIlraith, J. Christopher Beck, and Eric I. Hsu. 2012. Dsharp: Fast d-DNNF Compilation with sharpSAT. In *Advances in Artificial Intelligence*, Leila Kosseim and Diana Inkpen (Eds.). Springer, 356–361.

[53] Daniel-Jesus Munoz, Jeho Oh, Mónica Pinto, Lidia Fuentes, and Don Batory. 2019. Uniform Random Sampling Product Configurations of Feature Models That Have Numerical Features. In *Proc. Int'l Systems and Software Product Line Conf. (SPLC)*. ACM, 289–301.

[54] Jeho Oh, Paul Gazzillo, and Don Batory. 2019. t-wise Coverage by Uniform Sampling. In *Proc. Int'l Systems and Software Product Line Conf. (SPLC)*. ACM, 84–87.

[55] Jeho Oh, Paul Gazzillo, Don Batory, Marijn Heule, and Maggie Myers. 2019. *Uniform Sampling From Kconfig Feature Models*. Technical Report TR-19-02. University of Texas at Austin, Department of Computer Science.

[56] Héctor Palacios, Blai Bonet, Adnan Darwiche, and Hector Geffner. 2005. Pruning Conformant Plans by Counting Models on Compiled d-DNNF Representations. In *Proc. Int'l Conf. on Automated Planning and Scheduling (ICAPS)*, Vol. 5. AAAI Press, 141–150.

[57] Juliana Alves Pereira, Pawel Matuszyk, Sebastian Krieter, Myra Spiliopoulou, and Gunter Saake. 2016. A Feature-Based Personalized Recommender System for Product-Line Configuration. In *Proc. Int'l Conf. on Generative Programming: Concepts & Experiences (GPCE)*. ACM, 12 pages.

[58] Richard Pohl, Kim Lauenroth, and Klaus Pohl. 2011. A Performance Comparison of Contemporary Algorithmic Approaches for Automated Analysis Operations on Feature Models. In *Proc. Int'l Conf. on Automated Software Engineering (ASE)*. IEEE, 313–322.

[59] Tian Sang, Paul Beame, and Henry Kautz. 2005. Heuristics for Fast Exact Model Counting. In *Proc. Int'l Conf. on Theory and Applications of Satisfiability Testing (SAT)*. Springer, 226–240.
[60] Palash Sashittal and Mohammed El-Kebir. 2019. *SharpTNI: Counting and Sampling Parsimonious Transmission Networks under a Weak Bottleneck*. Technical Report.
[61] Sebastian Schlag, Tobias Heuer, Lars Gottesbüren, Yaroslav Akhremtsev, Christian Schulz, and Peter Sanders. 2023. High-Quality Hypergraph Partitioning. *ACM J. of Experimental Algorithmics (JEA)* 27 (2023), 1–39.
[62] Julia Schroeter, Malte Lochau, and Tim Winkelmann. 2012. Multi-Perspectives on Feature Models. In *Proc. Int'l Conf. on Model Driven Engineering Languages and Systems (MODELS)*. Springer, 252–268.
[63] Reimar Schröter, Sebastian Krieter, Thomas Thüm, Fabian Benduhn, and Gunter Saake. 2016. Feature-Model Interfaces: The Highway to Compositional Analyses of Highly-Configurable Systems. In *Proc. Int'l Conf. on Software Engineering (ICSE)*. ACM, 667–678.
[64] Reimar Schröter, Sebastian Krieter, Thomas Thüm, Fabian Benduhn, and Gunter Saake. 2017. Compositional Analyses of Highly-Configurable Systems With Feature-Model Interfaces. In *Proc. Software Engineering (SE)*, Jan Jürjens and Kurt Schneider (Eds.). Gesellschaft für Informatik, 129–130.
[65] Sergio Segura. 2008. Automated Analysis of Feature Models Using Atomic Sets. In *Proc. Int'l Systems and Software Product Line Conf. (SPLC)*, Vol. 2. IEEE, 201–207.
[66] Shubham Sharma, Rahul Gupta, Subhajit Roy, and Kuldeep S Meel. 2018. Knowledge Compilation Meets Uniform Sampling. In *Proc. Int'l Conf. on Logic for Programming, Artificial Intelligence, and Reasoning*. EasyChair, 620–636.
[67] Shubham Sharma, Subhajit Roy, Mate Soos, and Kuldeep S Meel. 2019. GANAK: A Scalable Probabilistic Exact Model Counter. In *Proc. Int'l Joint Conf. on Artificial Intelligence (IJCAI)*, Vol. 19. AAAI Press, 1169–1176.
[68] Sajjad Siddiqi and Jinbo Huang. 2008. Probabilistic Sequential Diagnosis by Compilation. In *Proc. Int'l Symposium on Artificial Intelligence and Mathematics (ISAIM)*. AAAI Press.
[69] Mate Soos and Kuldeep S. Meel. 2022. Arjun: An Efficient Independent Support Computation Technique and its Applications to Counting and Sampling. In *Proc. Int'l Conf. on Computer-Aided Design (ICCAD)*. ACM, Article 71.
[70] Chico Sundermann, Tobias Heß, Michael Nieke, Paul Maximilian Bittner, Jeffrey M. Young, Thomas Thüm, and Ina Schaefer. 2024. Evaluating State-of-the-Art #SAT Solvers on Industrial Configuration Spaces. In *Proc. Software Engineering (SE)*. Gesellschaft für Informatik. To appear.
[71] Chico Sundermann, Elias Kuiter, Tobias Heß, Heiko Raab, Sebastian Krieter, and Thomas Thüm. 2023. On the Benefits of Knowledge Compilation for Feature-Model Analyses. *Annals of Mathematics and Artificial Intelligence (AMAI)* (2023).
[72] Chico Sundermann, Michael Nieke, Paul Maximilian Bittner, Tobias Heß, Thomas Thüm, and Ina Schaefer. 2021. Applications of #SAT Solvers on Feature Models. In *Proc. Int'l Working Conf. on Variability Modelling of Software-Intensive Systems (VaMoS)*. ACM, Article 12, 10 pages.
[73] Chico Sundermann, Heiko Raab, Tobias Heß, Thomas Thüm, and Ina Schaefer. 2024. Reusing d-DNNFs for Efficient Feature-Model Counting. *Trans. on Software Engineering and Methodology (TOSEM)* (2024). To appear.
[74] Samuel Teuber and Alexander Weigl. 2021. Quantifying Software Reliability via Model-Counting. In *Proc. Int'l Conf. on Quantitative Evaluation of Systems (QEST)*. Springer, 59–79.
[75] Thomas Thüm, Christian Kästner, Sebastian Erdweg, and Norbert Siegmund. 2011. Abstract Features in Feature Modeling. In *Proc. Int'l Systems and Software Product Line Conf. (SPLC)*. IEEE, 191–200.
[76] Marc Thurley. 2006. sharpSAT - Counting Models With Advanced Component Caching and Implicit BCP. In *Proc. Int'l Conf. on Theory and Applications of Satisfiability Testing (SAT)*. Springer, 424–429.
[77] Erik Peter Zawadzki, André Platzer, and Geoffrey J Gordon. 2013. A Generalization of SAT and# SAT for Robust Policy Evaluation.. In *Proc. Int'l Joint Conf. on Artificial Intelligence (IJCAI)*. AAAI Press, 2583–2590.

C.8 TSE Submission: Tackling Expressive Feature-Modeling Constructs with Pseudo-Boolean d-DNNF Compilation

Note that the preprint referenced in this thesis [C4] has a slightly adapted name from the TSE submission attached below.

[C4] Chico Sundermann, Stefan Vill, Elias Kuiter, Sebastian Krieter, Thüm, and Matthias Tichy. *Pseudo-Boolean d-DNNF Compilation for Expressive Feature Modeling Constructs*. Tech. rep. arXiv:2505.05976. Submitted to TSE. Cornell University Library, 2025. DOI: 10.48550/arXiv.2505.05976

Preprint published under **CC-BY** (https://creativecommons.org/licenses/by/4.0/), under review and will be published under same license if accepted.

Tackling Expressive Feature-Modeling Constructs with Pseudo-Boolean d-DNNF Compilation

Chico Sundermann*†, Stefan Vill*, Elias Kuiter‡, Sebastian Krieter†, Thomas Thüm†, and Matthias Tichy*
* Ulm University, Germany † TU Braunschweig, Germany ‡ University of Magdeburg, Germany

***Abstract*—Configurable systems typically consist of reusable assets that have dependencies between each other. To specify such dependencies, feature models are commonly used. As feature models in practice are often complex, automated reasoning is typically employed to analyze the dependencies. Here, the de facto standard is translating the feature model to conjunctive normal form (CNF) to enable employing off-the-shelf tools, such as SAT or #SAT solvers. However, modern feature-modeling dialects often contain constructs, such as cardinality constraints, that are ill-suited for conversion to CNF. This mismatch between the input of reasoning engines and the available feature-modeling dialects limits the applicability of the more expressive constructs. In this work, we shorten this gap between expressive constructs and scalable automated reasoning. Our contribution is twofold: First, we provide a pseudo-Boolean encoding for feature models, which facilitates smaller representations of commonly employed constructs compared to Boolean encoding. Second, we propose a novel method to compile pseudo-Boolean formulas to Boolean d-DNNF. With the compiled d-DNNFs, we can resort to a plethora of efficient analyses already used in feature modeling. Our empirical evaluation shows that our proposal substantially outperforms the state-of-the-art based on CNF inputs for expressive constructs. For every considered dataset representing different feature models and feature-modeling constructs, the feature models can be significantly faster translated to pseudo-Boolean than to CNF. Overall, deriving d-DNNFs from a feature model with the targeted expressive constraints can be substantially accelerated using our pseudo-Boolean approach. For instance, the Boolean approach only scales for group cardinalities with up-to 15 features while pseudo-Boolean d-DNNF compilation can compile cardinalities with thousands of features. Furthermore, our approach is competitive on feature models with only basic constructs.**

***Index Terms*—product lines, knowledge compilation, d-DNNF**

I. Introduction

Most aspects of modern life, such as transportation [1]–[3], communication [4], [5], and computers [6], [7], rely on configurable systems. Such systems typically consist of multiple features that can be combined (i.e., configured) according to a set of constraints. Variability languages are used to specify such constraints between features [8], [9]. For instance, a constraint may specify that a feature depends on another feature or that at least one of a group of features needs to be selected. A common specification in variability languages are feature models [9], which define features and constraints (e.g., dependencies) between them [10].

Industrial systems often come with thousands of constraints [2], [6], [11], [12], which mandates the usage of automated analyses [3]. For instance, common analyses are checking whether a given configuration is valid or computing the number of valid configurations [13], [14]. The de facto standard for analyzing feature models is translating them to conjunctive normal form (CNF) then applying standardized reasoning engines, such as SAT [10], [13], [15] or #SAT solvers [2], [16], [17].

There is a mismatch between constraint types in available variability languages and constraints supported by existing reasoning engines. While research on feature-model analysis is mostly limited to Boolean logic [15], [18], the majority of variability languages support constraints that are hard to represent in CNF [9], [19], [20]. Commonly considered examples for such constructs are group cardinality (i.e., select between n and m features from a group), feature cardinality (i.e., select a single feature between n and m times), or constraints over numeric attributes (e.g., overall power usage of features is limited), or non-Boolean features [9], [21]–[23]. For those constructs, translating to Boolean logic may lead to an exponential increase in formula size [24]. In a survey of ter Beek et al. [9], 11 out 13 variability languages include at least one of such constructs. Consequentially, it is often infeasible to represent and reason about feature models containing those constraints with the standard Boolean approaches [24].

Such expressive constructs are also part of most languages used in practice. In a survey on variability-modeling practices in the industry of Berger et al. [25], pure variants [26] was the most used off-the-shelf tool. Their specification relies on various expressive constructs, including group cardinalities and feature attributes [23]. The tools following in popularity, namely Gears, FeatureIDE [27], [28] and DOPLER [29], also support more expressive constructs.

Several studies also report the mandate of these constructs in real-world projects [20], [24], [30]–[32]. Hubaux et al. [30] conducted a survey with four[1] companies and report that feature cardinalities and attribute constraints are desired. Furthermore, expressive constructs, including group cardinalities, attribute constraints, and feature cardinalities, were also observed in other real-world case studies, such as the embedded system eCos [20], configurable jewelry-rings [31], video conference systems by Cisco [32], and university courses [24].

Scalable solutions for various analyses on expressive constructs are lacking. While solutions for checking satisfiability are also available for more expressive formats (e.g., SMT [33], [34] or CP [35], [36]), such techniques have not been yet

[1]The four companies come from the computer hardware, meeting management, and document management application domain [30].

extended for exact counting or enumeration. Furthermore, the expressive power of these approaches also comes with added computational cost [33], [36], [37]. Both of these operations are relevant for a plethora of feature-model analyses [13], [16], [38]. Another issue hindering scalability is that many feature-model analyses rely on more than a single invocation of those complex computations [13]. Knowledge compilation is a strategy for automated reasoning which has recently attracted attention for feature-model analysis [39]–[41]. With knowledge compilation, the feature model is translated to a format (e.g., d-DNNF [42] or BDD [43]) with beneficial properties [44]. These formats can be expensive to compile initially, but after this one-time initial effort, they support more efficient analysis techniques [44]. However, compilation for formats popular in feature-model analysis is currently strictly limited to Boolean formulas [13], [42], [45]–[47], which excludes the more expressive feature-modeling constructs.

In this work, we aim to shorten the gap between expressive feature-modeling constructs and suitable reasoning engines. Our strategy to shorten this gap is twofold. First, we provide pseudo-Boolean encodings for common feature-modeling constructs which are often more natural and substantially smaller than respective Boolean encodings. In particular, we target constructs from the Universal Variability Language (UVL) [21], namely all basic constructs and group cardinalities, feature cardinalities, and attribute constraints. We use UVL as reference as it is widely employed [21], [28], [48]–[52] and covers many common constructs [9], [21]. The advantage of pseudo-Boolean formulas is that they allow numeric constants, while still only having to manage Boolean variables. Using this flexibility compared to CNFs facilitates encoding expressive constructs while profiting from the performance advantages of strictly Boolean variables. This allows us to reuse established strategies from Boolean [46], [53] and pseudo-Boolean reasoning [54], [55]. The idea is to find a middle ground between powerful[2] but consequentially more computationally demanding reasoning, such as SMT [33], [34], or CP [35]–[37], and purely Boolean formulas [15]. Second, we introduce pseudo-Boolean d-DNNF compilation, which enables compiling pseudo-Boolean constraints to a Boolean d-DNNF. The resulting d-DNNF can then be used to efficiently perform a plethora of relevant feature-model analyses [13], [39], [40], [56]. For instance, counting and enumeration configurations can be performed in linear time (w.r.t. d-DNNF size) on a d-DNNF [13], [44]. Overall, we provide the following contributions in this work:

- We present *pseudo-Boolean encodings* for several common feature-modeling constructs, including the expressive constructs group cardinality, feature cardinality and attribute constraints (Section III).
- We propose a core algorithm and optimizations for *compiling pseudo-Boolean constraints to d-DNNF* (Section IV).
- We provide *publicly available tooling* for both pseudo-Boolean encoding and pseudo-Boolean d-DNNF compilation.
- We examine the advantages of our approach regarding both encoding and compilation over the state-of-the-art with a *large-scale empirical evaluation* (Section V).

[2]Power with respect to the expressiveness of supported formulas.

II. Background

In this section, we introduce the background required for understanding the main concepts in this work, namely feature models, automated reasoning, d-DNNF compilation, and pseudo-Boolean logic.

A. Feature Models

Feature models are a commonly used formalism to specify the set of valid configurations for a configurable system [57]. Typically, a feature model consists of a feature hierarchy, denoting parent-child relationships with additional cross-tree constraints [14], [57]. Often, non-functional attributes can be attached to features (e.g., a price) [9], [21], [34], [58], [59]. However, the types of parent-child relationships and constraints that are supported vastly differ in literature [9].

Table I shows a taxonomy of different feature-modeling classes. The taxonomy is inspired by the language levels used in the Universal Variability Language (UVL) [21]. We use UVL as reference since UVL is widely used [28], [48]–[51] and covers many common constructs [9], [21]. Furthermore, UVL language levels and our taxonomy aim to categorize the feature-modelling based on complexity for reasoning engines. For instance, basic feature models can be translated to CNF within poly-time and poly-space [60], which is the de facto standard input for SAT [61], [62] or #SAT [53], [63], [64] solvers. In contrast, feature models with different feature types require more powerful reasoning (e.g., SMT [34], [48], [65]). Overall, we differentiate between four different classes, namely basic $\mathbb{B}$, cardinality $\mathbb{C}$, attribute constraints $\mathbb{A}$, and feature types $\mathbb{T}$. The classes differ in the supported group types (i.e., hierarchical parent-child relationships), types of cross-tree constraints (e.g., propositional), and feature types (e.g., Boolean or numeric). In this work, we consider all constructs from $\mathbb{C}$, $\mathbb{A}$, and $\mathbb{T}$ as *expressive* constructs. Furthermore, we use $G_{\mathbb{X}}$ to denote the group types introduced by the class $\mathbb{X}$ and $C_{\mathbb{X}}$ types of cross-tree constraints introduced by the class $\mathbb{X}$. For instance, the cardinality class $\mathbb{C}$ introduces the group types $G_{\mathbb{C}} = \{\text{group cardinality, feature cardinality}\}$.

Basic feature models are commonly considered, especially in the context of automated reasoning [13], [15], [27], [66], [67]. Figure 1 showcases a basic feature model specifying the valid configurations of a robot vacuum product line. Each `robot vacuum` requires some kind of `obstacle detection` and `extra storage` (*mandatory* flag). Furthermore, the robots may have `maps`, a `mop mode`, and a `camera` (*optional* flag). For the obstacle detection, users may choose between one or more from `sensor`, `AI`, and `physical` (*or* relation). The extra storage can be either a `dust storage` or a `water storage` (*alternative* relation). In addition to the hierarchy, two cross-tree constraints denote that `AI` and `maps` rely on having a `camera` and that `mop mode` requires a `water storage`. In basic feature models, the typical parent-child relationships are *optional*,

TABLE I
FEATURE-MODELING CONSTRUCT TAXONOMY INSPIRED BY UVL LANGUAGE LEVELS [21]

Class	Introduced Groups ($G_{\mathbb{X}}$)	Introduced Cross-tree Constraints ($C_{\mathbb{X}}$)	Introduced Feature Types
Basic $\mathbb{B}$	{Alternative, or, mandatory, optional}	Propositional ($\wedge, \vee, \neg, \Leftrightarrow, \Rightarrow$)	{Boolean}
Cardinality $\mathbb{C}$	{group cardinality, feature cardinality}	-	
Attribute Constraints $\mathbb{A}$	-	attribute terms	-
Feature Types $\mathbb{T}$	-	feature terms	{numeric, string}

mandatory, *alternative*, and *or* [21], [27], [66]. Furthermore, cross-tree constraints are limited to propositional logic with common operators.

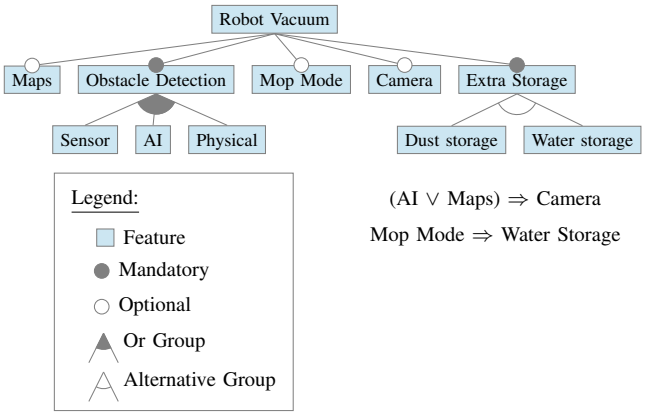


Fig. 1. Basic Feature Model

However, the majority of available variability languages also support at least some more expressive constructs, such as cardinalities, attributes and constraints over them, or typed features [9], [22], [23]. Figure 2 shows an extension of the feature model shown in Figure 1 with constructs from $\mathbb{B}$, $\mathbb{C}$, and $\mathbb{A}$. For obstacle detection, due to a limited number of connections, only one or two features can be selected (*group cardinality* [1..2]). Between one and three extra storage components can now be configured (feature cardinality [1..3]). In addition, the features now have non-functional properties, called *feature attributes* [58]. Here, most features have a cost and the storage components also have an attribute indicating the space they require. These attributes can be used for informational purposes or to specify constraints over them. In our example, the overall cost is limited to 15 and the overall space for storages is limited to 6. Various interpretations of feature attribute semantics have been discussed [21], [34], [68]–[70]. We follow UVL [21] regarding the semantics of feature attributes. There is one important difference between feature attributes and typed features: attributes cannot be configured individually (i.e., their value is constant when deriving configurations) [21], [58] . Hence, features are variables while attributes are constants which are typically easier to handle for reasoning engines. Formally, we consider attributes $a_i = (f_i, v_i)$ as always attached to a feature f_i. In a configuration, their value is v_i if its feature f_i is selected and zero otherwise. We discuss the semantics of the presented relationships and constraints in Section III. Nevertheless, we refer to related work on the structure and semantics of feature models for more details [10], [21], [57].

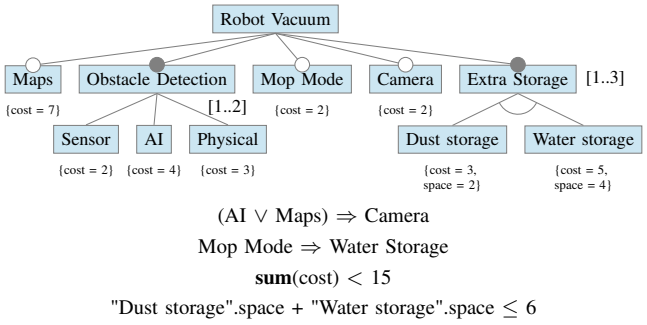


Fig. 2. Feature Model Including Expressive Constructs

B. Logic-Based Reasoning

The de facto standard for feature-model analysis is translating the feature model to Boolean logic [10], [13]–[15], [71], [72] and then invoke off-the-shelf solvers [27], [53], [62], [66], [73]. Here, the features are mapped to Boolean variables and the parent-child relationships and cross-tree constraints translated to Boolean formulas, typically in conjunctive normal form (CNF) [74]. A CNF C is a single conjunction of clauses $C = (C_1 \wedge \ldots \wedge C_n)$. Each clause C_i is a disjunction of literals $C_i = (l_{i,1} \vee \ldots \vee l_{i,m})$. For basic feature models $\mathbb{B}$, translations to CNF are well known and commonly employed [27], [60], [75]. However, constructs from the expressive classes $\mathbb{C}$, $\mathbb{A}$, and $\mathbb{T}$ are not straightforward to translate to CNF. In general, there is a lack of scalable automated reasoning techniques for those expressive constructs.

C. d-DNNF Compilation

Knowledge compilation refers to compiling an original formula (e.g., representing a feature model) to another format with beneficial properties regarding runtime of complex operations [44]. There are various knowledge-compilation target languages that each offer varying tradeoffs between (1) the complexity to compile and (2) operations that are possible in polynomial time on the compiled artifact [44]. Typical operations are checking satisfiability, model counting, enumerating solutions, or algebraic operations [13], [44].

The *deterministic decomposable negation normal form* (d-DNNF) is a knowledge compilation target language that has gained traction in feature-model analysis over the past years [39], [40], [56], [76]. d-DNNFs support model counting, checking satisfiability, and enumerating solutions within linear

time with respect to the d-DNNF's size [44], which can be used to accelerate a plethora of feature-model analyses [13]. None of these operations can be computed in polynomial time on a CNF [44]. The linear runtime operations for d-DNNFs are enabled by the properties *determinism* and *decomposability*. A formula is deterministic if all disjuncts d_i in a disjunction $D = d_1 \vee \ldots d_n$ are pairwise logically contradicting (i.e., $\forall i, j : i \neq j : d_i \wedge d_j \equiv \bot$). A formula is decomposable if no conjunct c_i in a conjunction $C = c_1 \ldots c_n$ shares variables with another conjunct (i.e., $\forall i, j : i \neq j : vars(c_i) \cup vars(c_j) = \emptyset$).

Modern d-DNNF compilation is based on the exhaustive Davis-Putnam-Logemann-Loveland (DPLL) algorithm [42], [45], [46]. Here, the solution space of the formula is explored by recursively assigning variables until every solution is found. Algorithm 1 shows the base procedure to compile a given CNF to d-DNNF. There are two main aspects to consider. First, while the formula is not yet satisfied nor unsatisfied, variables are assigned recursively. The subformulas $F|_v$ and $F|_{\neg v}$ resulting from setting v to true and false are recursively compiled to d-DNNF and combined to a disjunction $(F|_v \wedge v) \vee (F|_{\neg v} \wedge \neg v)$. Note that this disjunction is equivalent to F and per construction deterministic as every leftside solution contains v and every rightside solution contains $\neg v$. Second, if the formula can be split into subformulas that share no variables (i.e., components), those components are then processed separately. The compiled sub d-DNNFs can then be conjuncted. Note that the conjunction is equivalent to F and decomposable by construction. Following this procedure, we acquire a semantically equivalent d-DNNF since every conjunction is decomposable and every disjunction is deterministic.

Algorithm 1 Boolean d-DNNF Compilation

```
1: Input: CNF F
2: Output: d-DNNF D
3: Procedure: compile(F)
4: if satisfied(F) then
5:     return ⊤
6: else if !satisfied(F) then
7:     return ⊥
8: end if
9: components ← identifyDisconnectedComponents(F)
10: subs ← [ ]
11: for C : components do
12:     if sat(C) then
13:         continue
14:     end if
15:     subs.push(compile(F|_v ∧ v) ∨ compile(F|_¬v ∧ ¬v))
16: end for
17: return ⋀_{S∈subs} S
```

D. Pseudo-Boolean Logic

A linear pseudo-Boolean constraint is an inequality following the structure shown in Equation 1 [54], [77]. Here, k_i and b are constants while x_i are Boolean variables with $x_i \rightarrow \{0, 1\}$, where 0 corresponds to false and 1 to true. The set vars(F) denotes the variables considered in the formula. The constants k_i are called *factors* and the rightside b is called *degree* [54]. Each pseudo-Boolean formula is a single conjunction over such constraints. Consequentially, pseudo-Boolean formulas are a generalization of CNFs. A CNF clause $(x_1 \vee \ldots \vee x_n)$ can be represented as $\sum_{i=1}^{n} 1 \cdot x_i \geq 1$.

$$\sum_{i=1}^{n} k_i \cdot x_i \geq b \text{ with } k_i, b \in \mathbb{Z}, x_i \in \text{vars}(F) \tag{1}$$

The advantage of pseudo-Boolean is the added expressiveness over CNFs with numerical constants while preserving strictly Boolean variables [54], [77]. With the added expressiveness, certain constructs can be encoded more sparsely (i.e., with fewer literals). In Section III, we showcase these advantages by encoding different feature-modeling constructs in pseudo-Boolean logic. By having strictly Boolean variables, we can reuse strategies (cf. Section IV) that have been optimized over decades of research on Boolean [46], [53], [61] and pseudo-Boolean [54], [55], [78] reasoning.

III. Pseudo-Boolean Feature Model Encoding

To use the benefits of pseudo-Boolean logic for feature modeling, we first need encodings of commonly occuring constructs. While Boolean encodings for feature models are well researched, we only found a Bachelor's thesis by Hennerberg [79] addressing encodings as pseudo-Boolean formulas, but they are limited to basic constructs, group cardinalities, and sums over attributes. Furthermore, their group cardinality encoding produces faulty results if the parent is deselected [79]. Hence, we provide an extended collection of pseudo-Boolean encodings. Note that there are typically multiple valid encodings for a feature-modeling construct. In this work, for each Boolean and pseudo-Boolean, we limit ourselves to one *semantically equivalent* encoding per construct.

Various feature-modeling constructs have been considered with varying degrees of adoption in practice [9]. With our approach, we limit ourselves to constructs from the Universal Variability Language (UVL) [21] for now. We use UVL as reference since UVL is widely used [28], [48]–[51] and covers many common constructs [21]. Furthermore, the design decisions including supported constructs are result of a community survey with researchers and practitioners [80] which increases the chances for the constructs to be relevant. With pseudo-Boolean encoding, we aim to find a sweet spot that provides more efficient encodings for several constructs compared to CNF (cf. Section III-A and Section III-B), but can still capitalize on the performance advantages of purely Boolean variables. Hence, we target the first three classes of our taxonomy, namely basic $\mathbb{B}$, cardinality $\mathbb{C}$, and attribute constraints $\mathbb{A}$. The taxonomy classes $\mathbb{B}$, $\mathbb{C}$, and $\mathbb{A}$ cover all UVL language levels but *typed* features [21].

A. Encoding Basic Feature Models

The goal of employing pseudo-Boolean logic for feature models is mainly to provide more efficient encodings

(w.r.t. size and translation runtime) for additional expressive constructs. Nevertheless, to translate feature models in practice we initially require encodings for basic constructs. Table II shows Boolean and pseudo-Boolean encodings for the hierarchy group types in features models from the basic class $\mathbb{B}$. Here, $grp^p_{c_1..c_n}$ describes the relationship $grp \in \{opt, mand, or, alt\}$ between parent p and its children $\{c_1, \ldots, c_n\}$. Note that a feature c_i can only be child feature in one group, but a parent feature p may have multiple groups. For the Boolean encoding, we use FeatureIDE's[3] [27] translations as reference, as they are commonly used [50], [56], [60], [81] and in line with other typical definitions from the literature. For the pseudo-Boolean encoding, our translations are similar to the ones provided by Henneberg [79].

TABLE II
ENCODINGS FOR BASIC FEATURE MODEL HIERARCHY

Constraint	Boolean	Pseudo-Boolean
$opt^p_{c_1..c_n}$	$\bigwedge_{i=1}^n (c_i \Rightarrow p)$	$n \cdot p + \sum_{i=1}^n -c_i \geq 0$
$mand^p_{c_1..c_n}$	$\bigwedge_{i=1}^n (c_i \Leftrightarrow p)$	$n \cdot p + \sum_{i=1}^n -c_i = 0$
$or^p_{c_1..c_n}$	$p \Leftrightarrow (\bigvee_{i=1}^n c_i)$	$n \cdot p + \sum_{i=1}^n -c_i \geq 0$ $-p + \sum_{i=1}^n c_i \geq 0$
$alt^p_{c_1..c_n}$	$p \Leftrightarrow (\bigvee_{i=1}^n c_i \wedge \bigwedge_{i<j} (\neg c_i \vee \neg c_j))$	$p + \sum_{i=1}^n -c_i = 0$

a) Optional Features: For *optional* features, no constraints are imposed on the child features when the parent is selected. Hence, an optional group only enforces that the selection of a child mandates the selection of the parent feature. Table II shows the Boolean and pseudo-Boolean encoding for an optional group $opt^p_{c_1..c_n}$. The pseudo-Boolean constraint is only violated (i.e., left side ≥ 0) if a child is selected but the parent is not. Note that $n \cdot p$ is a valid summand in a pseudo-Boolean constraint as n is a constant and p is a variable.

b) Mandatory Features: Mandatory features need to be selected if the parent is selected. The pseudo-Boolean constraint for $mand^p_{c_1..c_n}$ given in Table II is violated if the parent but not all its children are selected. Note that the provided pseudo-Boolean constraint does not exactly match the definition from Section II-D (i.e., $=$ instead of $\geq$), but can be straightforwardly normalized to the desired form. We discuss normalizations we perform at the end of this section.

c) Or Group: If the parent of an *or* group $or^p_{c_1,..c_n}$, is selected, at least one of its child features need to be selected. To specify $or^p_{c_1,..c_n}$, we use two pseudo-Boolean constraints that ensure (1) no c_i is selected while p is not and (2) selecting p requires selecting at least one c_i.

d) Alternative Group: If the parent of an *alternative group* $alt^p_{c_1..c_n}$ is selected, exactly one of its child features needs to be selected. The pseudo-Boolean constraint for $alt^p_{c_1..c_n}$ is only satisfied if the parent p and exactly one child c_i (i.e., $1 - 1 = 0$) is selected or no parent and no child (i.e., $0 - 0 = 0$) is selected.

e) Boolean Cross-Tree Constraints: To translate Boolean cross-tree constraints to pseudo-Boolean logic, we can use well-known translations to CNF and then translate the resulting clauses to pseudo-Boolean constraints. In particular, we convert a clause $(x_1 \vee \ldots \vee x_n)$ to the pseudo-Boolean constraint $\sum_{i=1}^n 1 \cdot x_i \geq 1$. The benefit is that we employ available techniques and tools for CNF translation [60].

By combining these translations, we can represent basic feature models in pseudo-Boolean logic. For basic feature models, most constructs can be represented as formulas with similar sizes in Boolean and pseudo-Boolean. The exception are alternatives $alt^p_{c_1..c_n}$ which induce considerable difference as we require $\mathcal{O}(n^2)$ clauses in a CNF, but only a single pseudo-Boolean constraint.

B. Encoding Expressive Constraints

Table III shows Boolean and pseudo-Boolean encodings for constructs in the taxonomy classes cardinality $\mathbb{C}$ and attribute constraints $\mathbb{A}$. As reference for the Boolean encoding, we use UVL conversion strategies [21], [82] which translate the expressive constructs to propositional constraints. We employ the UVL conversion strategies as (1) they are part of the standard UVL tooling and (2) each conversion produces semantically equivalent formulas [21], [82].

TABLE III
ENCODINGS FOR EXPRESSIVE FEATURE-MODELING CONSTRUCTS

Construct	Boolean	Pseudo-Boolean
$[a..b]^p_{c_1..c_n}$	enum($[a..b]^p_{c_1..c_n}$)	$n \cdot p + \sum_{i=1}^n -c_i \geq 0$ $-a \cdot p + \sum_{i=1}^n c_i \geq 0$ $\sum_{i=1}^n -c_i \geq -b$
$f[a..b]$	enum($[a..b]^{cr}_{c_1,..,c_b}$)	$pb([a..b]^{cr}_{c_1,..,c_b})$
$\phi(A) \star \psi(A')$	enum($\phi(A) \star \psi(A')$)	Table IV

a) Group Cardinality: A group cardinality $[a..b]^p_{c_1..c_n}$ specifies that if parent p is selected, between a and b of its children c_i need to be selected. For Boolean, we use the UVL conversion strategy which enumerates the valid assignments over the children $c_1..c_n$ that satisfy the group cardinality. For instance, $[2..3]^p_{\{c_1,c_2,c_3\}}$ would be encoded as $(c_1 \wedge c_2 \wedge c_3) \vee (c_1 \wedge \bar{c_2} \wedge \bar{c_3}) \vee (\bar{c_1} \wedge c_2 \wedge \bar{c_3}) \vee (\bar{c_1} \wedge \bar{c_2} \wedge c_3)$. Note that several alternative encodings for at-least-k and at-most-k constraints have been suggested in the literature [24], [83]–[85]. We discuss those encodings and their suitability for d-DNNF compilation in Section VI. The enumeration strategy is also employed by UVL for the remaining expressive constructs [21]. With our pseudo-Boolean encoding shown in the first rows of Table III, we use three constraints that (1) ensure that child features $c_1, .., c_n$ can only be selected if parent p is selected, (2) if p is selected, at least a child features are selected, and (3) at most b child features are selected. Henneberg [79] also provides an encoding for group cardinalities, but his proposal may produce incorrect results if the parent is deselected.

b) Feature Cardinality: A feature cardinality $f[a..b]$ specifies that a feature f may be selected between a and b times [21], [86]. It is important to consider that for every incarnation $f_{i \in [1,b]}$ of f, the subtree spanned by ancestors of f needs to be cloned as every incarnation f_i can be configured

[3]https://github.com/FeatureIDE/FeatureIDE. FeatureIDE is a widely used framework for software-product line development [27].

individually [21], [86], [87]. A feature cardinality can therefore be considered as group cardinality $[a..b]^{cr}_{f_1,..,f_b}$ over these clones $f_{i\in[1,b]}$ with a new artificial cardinality root feature cr. This semantically equivalent interpretation is implicitly used by UVL [21] and explicitly used for our pseudo-Boolean encoding. We use the previously presented strategy to convert the resulting group cardinality ($pb([a..b]^{cr}_{c_1,..,c_b})$ in Table III). We manage cross-tree constraints containing cloned variables in the same way as UVL [21].

c) Numeric Attribute Constraints: In UVL, numeric attribute constraints are terms $\phi(A) \star \psi(A')$ over constants and a set of feature attributes A with $\star \in \{=, \neq, \geq, \leq, >, <\}$. A term $\phi(A)$ is an arbitrary expression over attributes $a_i \in A$ and arithmetic operations (i.e., $+, -, \div, \cdot$). An example is shown in Figure 2 where the space of storage units is limited ("Dust storage".space + "Water storage".space ≤ 6). In that example, we have $\phi(A) =$ ("Dust storage".space + "Water storage".space), $\psi(A') = 6$, and $\star = \leq$. Attributes $a_i = (f_i, v_i)$ resolve to their value v_i if the corresponding feature f_i is selected and to 0 if f_i is deselected. UVL also contains two aggregate functions: sum() and avg(). However, since both can be expanded (e.g., sum(a) to a sum over all instances a_i of that attribute), we do not explicitly consider them here. For the Boolean encoding, we also use the UVL conversion strategies, which enumerate satisfiable assignments over the features f_i corresponding to the respective attributes $a_i \in A \cup A'$ [21]. For the pseudo-Boolean encoding, we propose an approach that *recursively transforms subterms* to pseudo-Boolean logic depending on the connecting operator. We show the different transformations in Table IV.

TABLE IV
PSEUDO-BOOLEAN EXPRESSION ENCODINGS

Term	Target Encoding
c_i	c_i
$a_i = (v_i, f_i)$	$v_i \cdot f_i$
$(c_1 + \sum_{i=1}^{n} k_i \cdot x_i) +$ $(c_2 + \sum_{i=1}^{m} l_i \cdot y_i)$	$(c_1 + c_2) + \sum_{i=1}^{n} k_i \cdot x_i$ $+ \sum_{i=1}^{m} l_i \cdot y_i$
$(c_1 + \sum_{i=1}^{n} k_i \cdot x_i) -$ $(c_2 + \sum_{i=1}^{m} l_i \cdot y_i)$	$(c_1 - c_2) + \sum_{i=1}^{n} k_i \cdot x_i$ $+ \sum_{i=1}^{m} -l_i \cdot y_i$
$(c_1 + \sum_{i=1}^{n} k_i \cdot x_i) \cdot$ $(c_2 + \sum_{i=1}^{m} l_i \cdot y_i)$	$(c_1 \cdot c_2) + \sum_{i=1}^{n} c_2 \cdot k_i \cdot x_i$ $+ \sum_{i=1}^{n} \sum_{j=1}^{m} k_i \cdot l_j \cdot z_{i,j}$ $+ \sum_{i=1}^{m} c_1 \cdot l_i \cdot y_i$
$(c_1 + \sum_{i=1}^{n} k_i \cdot x_i) \div$ $(c_2 + \sum_{i=1}^{m} l_i \cdot y_i)$	$\sum_{s\in S} \frac{c_1}{c_2 + \sum_{i\in s} l_i} \cdot e_s$ $+ \sum_{j=1}^{n} \sum_{s\in S} \frac{k_j}{c_2 + \sum_{i\in s} l_i} \cdot e_s$

For unary terms, *constants* c_i and *feature attributes* a_i, the transformation is straightforward. To translate the binary expressions, we always assume that both sides are already in pseudo-Boolean format (i.e., $(c_1 + \sum_{i=1}^{n} k_i \cdot x_i)$).[4] Otherwise, we can just recursively translate them using the same procedure. For *addition* and *subtraction*, joining the sums and constants already produces a valid pseudo-Boolean expression. When resolving the *multiplication* of both sums, we produce $\sum_{i=1}^{n} \sum_{j=1}^{m} k_i \cdot l_j \cdot x_i \cdot y_j$, which is not in pseudo-Boolean format since x_i and y_j are both variables. Thus, we use a substitution variable $z_{i,j}$ and an additional constraint $z_{i,j} \Leftrightarrow x_i \wedge y_j$ for every pair (x_i, y_j). The added constraint ensures that $z_{i,j}$ behaves equivalently to $x_i \cdot y_i$. For a *division* of two sums, our encoding depends on enumerating the variable selections of the divisor. The main problem lies in the following partial construct after resolving the division: $\frac{k_i \cdot x_i}{c_2 + \sum_{i=1}^{m} l_i \cdot y_i}$. To get the required structure, we create a sum with each element representing the fraction under one variable selection $s \in S$: $\sum_{s\in S} \frac{k_i}{c_2 + \sum_{i\in s} l_i} \cdot e_s$. Each substitution variable e_s is equivalent to the corresponding selection s (i.e., $e_s \Leftrightarrow x_i \wedge \bigwedge_{i\in s} y_i \bigwedge_{\bar{i}\in s} \neg y_i$). In addition to that, we create pseudo-Boolean constraints that mark each assignment causing division by zero as unsatisfied (i.e., $s \Leftrightarrow \sum_{i=1}^{n} l_i \cdot y_i \neq -c_2$). Both our encodings for multiplication and division require the introduction of artificial variables, but the encodings preserve quasi-equivalence [60]. Hence, when compiling the resulting formula to d-DNNF later, the d-DNNF produces consistent results.

d) Normalizing Pseudo-Boolean Constraints: Some provided encodings slightly deviate from the structure of pseudo-Boolean constraints presented in Section II-D. The definition enforces $\geq$ as comparison operator, but our encodings also use $=$ and attribute constraints in UVL supports all the common comparison operators (i.e., $=$, $\geq$, $\leq$, $<$, $>$, and $\neq$). For all operators but $\neq$, we use the conversions shown in Table V to conform with the definition. In contrast, we keep $\neq$ constraints and handle them in our compiler as adapting them to match the structure in Equation 1 is complex. Furthermore, we convert floats to integers by multiplying both sides of the inequality with 10^X where x is the highest number of decimal places [54].

TABLE V
CONVERTING COMPARISON OPERATORS

Origin	Normalized
$\sum_{i=1}^{n} a_i \cdot v_i \leq b$	$\sum_{i=1}^{n} -a_i \cdot v_i \geq -b$
$\sum_{i=1}^{n} a_i \cdot v_i > b$	$\sum_{i=1}^{n} a_i \cdot v_i \geq b + 1$
$\sum_{i=1}^{n} a_i \cdot v_i < b$	$\sum_{i=1}^{n} -a_i \cdot v_i \geq -b + 1$
$\sum_{i=1}^{n} a_i \cdot v_i = b$	$\sum_{i=1}^{n} -a_i \cdot v_i \geq -b \wedge \sum_{i=1}^{n} a_i \cdot v_i \geq b$

e) Summary: By combining the different translation strategies, we can now encode all constructs from feature models with basic constructs $\mathbb{B}$, cardinality constructs $\mathbb{C}$, and attribute constraints $\mathbb{A}$ in pseudo-Boolean logic. For various constructs, the proposed pseudo-Boolean encoding is more sparse than the respective Boolean encoding. The differences are especially apparent for cardinalities and addition/subtraction based attribute constraints, where we achieve linear-sized encodings in pseudo-Boolean, but the Boolean encoding grows exponentially in size. Note that we examine the correctness and efficiency in practice of our encodings empirically in Section V. Having the encoding enables us to employ pseudo-Boolean based reasoning on UVL models, including our proposal; *pseudo-Boolean d-DNNF compilation* (cf. Section IV).

[4]The constraint does not conflict with the structure shown in Equation 1 as the additional constant c_1 can bet set of against the degree.

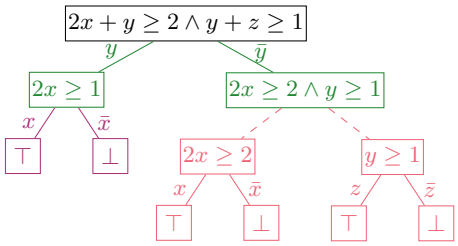


Fig. 3. Trace for Exhaustive Pseudo-Boolean DPLL

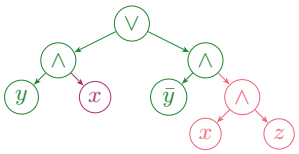


Fig. 4. d-DNNF Corresponding to Traversal in Figure 3

IV. Pseudo-Boolean d-DNNF Compilation

Our goal is to provide scalable reasoning for common feature-modeling constructs that are hard to analyze with state-of-the-art reasoning engines which typically use CNFs as input [15], [61], [62], [88], [89]. To this end, we propose pseudo-Boolean d-DNNF compilation, which comes with two main advantages. First, while the input is *pseudo-Boolean*, our approach produces *Boolean* d-DNNFs which can be used with available strategies that enable a plethora of feature-model analyses [39], [40], [56]. Second, pseudo-Boolean formulas can provide more sparse encodings for several constructs compared to CNF, while still only having to manage Boolean variables. The limitation to Boolean variables enables us to employ popular techniques from Boolean d-DNNF compilation with only slight adaptations. For instance, exhaustive DPLL [90], which is the algorithmic basis of every popular Boolean d-DNNF compiler [42], [45], [46], is not applicable for non-Boolean variables.

A. Reusing Ideas from Boolean Compilation

We propose pseudo-Boolean compilation based on exhaustive DPLL [90] which is also the de facto standard for Boolean compilation (cf. Section II-C) [42], [45], [46]. On the abstraction level of Algorithm 1, the algorithms for pseudo-Boolean and Boolean (i.e., using CNFs as input) are equivalent. However, many details need to be considered and changed for pseudo-Boolean d-DNNF compilation.

Figure 3 shows an example for our adaptation of exhaustive DPLL to a pseudo-Boolean formula. The adaptation follows previous adaptations of *non-exhaustive* DPLL for pseudo-Boolean formulas [54]. We recursively branch on the variables x, y, z until we have a conflict (i.e., at least one constraint cannot be satisfied by extending the current assignment) or all constraints are satisfied. Each time we assign a variable, we have to update every constraint that contains the assigned variable. For instance, $2x+y \geq 2 \wedge y+z \geq 1$ becomes $2x \geq 1$ under the assignment $y = 1$. When we reach a subformula, where we can split the constraints in a way that they share no variables, we traverse those splits separately as seen in Figure 3 for $2x \geq 2 \wedge z \geq 1$.

As the computed trace (Figure 3) only contains Boolean decisions, we can, analogously to Boolean compilation, derive a d-DNNF from it. Figure 4 shows the resulting d-DNNF. So, we get semantically equivalent results to the CNF-based state-of-the-art [46], but benefit from smaller input formulas for several feature-modeling constructs. For every assignment, we create a disjunction, which is, by construction, deterministic. Every time we split the formula due to disconnected components, we create a conjunction which is decomposable by construction. Note that we already propagated the $\top$ and $\bot$ nodes from Figure 3. For instance, we simplified $(x \wedge \top) \vee (\bar{x} \wedge \bot)$ to x.

B. Optimizations

The scalability of DPLL-based reasoning engines heavily relies on optimizations [46], [53], [91]–[93]. Over the last decades, various strategies for Boolean reasoning (e.g., SAT solving, model counting, and knowledge compilation) have been proposed. In the following, we briefly introduce some popular strategies that we have adopted for our pseudo-Boolean approach. Furthermore, we describe the required adaptations.

a) Boolean Constraint Propagation: Boolean constraint propagation refers to automatically assigning variables for which the formula is only satisfiable under that specific assignment. In Boolean logic, this is detected via unit clauses (i.e., clauses with exactly one literal). To satisfy the clause, the corresponding variable has to be assigned such that the literal is satisfied. For pseudo-Boolean logic, we use an adaptation suggested by Chai and Kuehlmann [54] to detect implied literals. Given $\sum_{i=1}^{n} a_i \cdot v_i \geq b$, it can be checked if the largest constant a_{max} is required to satisfy the inequality by checking $\sum_{i=1}^{n} a_i \cdot v_i < b + a_{max}$. In this case, v_{max} is implied.

b) Component Caching: During DPLL traversal, the algorithm often produces the same component (i.e., subformula) multiple times. With component caching, the goal is to save results of components to reuse them later [53], [94]. The main problem here is to efficiently store and detect components, which is typically realized via hashing. Popular hashing schemes used in Boolean solvers are not directly applicable for pseudo-Boolean formulas, because they rely on the structure of CNFs. However, hashing the entire data structure of a subformula has a substantial impact on the performance. We adapt the scheme used by sharpSAT [53] to also include the right side of the pseudo-Boolean inequalities. In particular, we encode a component as $((v_1, ..v_n)(l, r_l)(k, r_k))$ to prepare for hashing where v_i are unassigned variables, l and k are indices of unsatisfied constraints, and r_l and r_k their respective righthand sides. The lists are consistently sorted over the traversal. For instance, we would encode the subformula $2x \geq 2 \wedge y \geq 1$ resulting from setting $\bar{y}$ as $((x), (1, 2), (2, 1))$, where the left side of the second and third tuple refers to the index of the original constraint.

c) Conflict Analysis: In conflict analysis, the partial assignment leading to a conflict is saved as a conflict clause to prevent running into the same conflict multiple times [94].

For instance, after finding that the assignment $\{y, \bar{x}\}$ is unsatisfiable, we could add its negation as conflict clause (i.e., $\neg(y \wedge \bar{x}) \equiv (\bar{y} \vee x)$) to detect this conflict earlier during further branching. Common techniques aim to identify conflict clauses with as few as literals as possible as those can be potentially applied more often [94], [95]. Since assignments are still Boolean in pseudo-Boolean logic and clauses can be easily encoded as pseudo-Boolean constraints, we can use Boolean conflict-driven clause learning (CDCL) [94], [95] with only minor technical adaptations.

d) Variable Orderings: The ordering of variables during branching can have substantial impact on the compilation performance (e.g., regarding runtime) [42], [46]. Various strategies have been proposed that typically rely on the occurrence of variables in regular clauses (e.g., DLCS [96]) and conflict clauses (e.g., VSIDS [88]). For our compiler, we use the same primary variable sorting as `d4` [46]: prioritizing variables that result in disconnected components. In previous work, we observed that `d4` performs best for compiling feature models [13], [97]. Analogous to `d4`, we translate our formula to a hypergraph and use an existing partitioner to derive a cut set, which we then use for ordering. Within the cut set, we order the variables using an adaptation of VSIDS [88] which is commonly used in SAT and #SAT. Our adapted version of VSIDS also considers the impact of a variable in a pseudo-Boolean constraint considering the degree of the formula and the factor of the variable (cf. Section II-D). Furthermore, following insights from preliminary experiments, we use no no decay which would decrease the score of variables that are not part of conflict clauses over time [88].

C. Tooling

We provide publicly available tooling for both major steps of our pipeline, namely encoding feature models in pseudo-Boolean logic and pseudo-Boolean d-DNNF compilation. The encoder `UVL2pb`[5] is implemented in Java, so it can be directly used with the UVL parser.[6] The output `.opb` (standard format for pseudo-Boolean formulas) file can then be used as input for our compiler `p2d`[7], whose name is a tribute to the d-DNNF compiler `c2d` that compiles CNFs to d-DNNF. The pseudo-Boolean compiler `p2d` can be used for d-DNNF compilation and for model counting. The compiled d-DNNFs can be saved in the format proposed by `d4` [46], which can then be used as input for further reasoning (e.g., `ddnnife` [39] or `Winston` [56]).

V. Evaluation

With our empirical evaluation, we examine the benefits of our approach compared to the state-of-the-art. To this end, we compare the performance of pseudo-Boolean with Boolean d-DNNF compilation, including our pseudo-Boolean encoding of the feature model. For the performance, we consider representation size and runtime for both encoding and d-DNNF compilation. We provide a reproduction package including the evaluated tools, subject systems, and final results.[8]

A. Experiment Design

a) Research Questions: To examine the advantages of our approach over the state-of-the-art, we evaluate the performance of the *encoding* (**RQ1**) and the performance of the *compilation* (**RQ2**, **RQ3**).

RQ1 How does the performance of pseudo-Boolean encoding compare to CNF encoding for feature models?

RQ2 How does the performance of pseudo-Boolean d-DNNF compilation compare to Boolean compilation for industrial basic feature models?

RQ3 How does the performance of pseudo-Boolean d-DNNF compilation compare to Boolean compilation for expressive constructs?

For the performance, we consider two dimensions: size of representation w.r.t. to number of literals and runtime for compiling/encoding the input. With **RQ1**, we compare the runtimes required to encode the feature models in Boolean and pseudo-Boolean and the sizes of the resulting formulas. In **RQ2**, we examine the performance of our approach on feature models with only basic constructs for two reasons. First, while those constructs are used in practice [20], [22]–[24], [29]–[32], there are currently no industrial feature models with expressive constructs *publicly* available. Second, as basic feature-modeling constructs are widely used across systems and application domains [6], [8], [12], [98], [99], we expect that many feature models with more expressive constructs still contain a considerable share of basic constructs. Hence, the performance on basic models is still relevant for feature models containing expressive constructs. Even though there is a lack of publicly available feature models with the expressive constructs, we aim to provide indicative insights on how our approach performs for expressive constructs. With **RQ3**, we compare the performance of the pseudo-Boolean and Boolean approach for compiling feature models representing various combinations of constructs.

b) Evaluated Tools: With the evaluation, we aim to compare our approach to the Boolean state-of-the-art which is CNF to d-DNNF compilation [13], [46]. Figure 5 provides an overview on the pseudo-Boolean and Boolean pipeline. Our contributions are indicated with green arrows. For our approach, we evaluate our new proposals, namely `UVL2pb` for the pseudo-Boolean encoding and `p2d` for the d-DNNF compilation. For the Boolean baseline, we need a way to translate the models with expressive constructs to CNF. To this end, we use UVL conversion strategies [82] to translate the expressive models to basic models. Then, we use FeatureIDE's CNF translation, as it supports conversion from UVL and is commonly employed in other evaluations [39], [50], [56], [60], [81]. For Boolean d-DNNF compilation, we use `d4`[9] as other evaluations clearly indicate that `d4` performs best for feature-model analyis [39], [97].

[5] https://github.com/TUBS-ISF/pseudo-boolean-uvl-encoder
[6] https://github.com/Universal-Variability-Language/uvl-parser
[7] https://github.com/TUBS-ISF/p2d
[8] https://github.com/TUBS-ISF/pb-ddnnf-eval (Zenodo for final version)
[9] https://github.com/crillab/d4v2

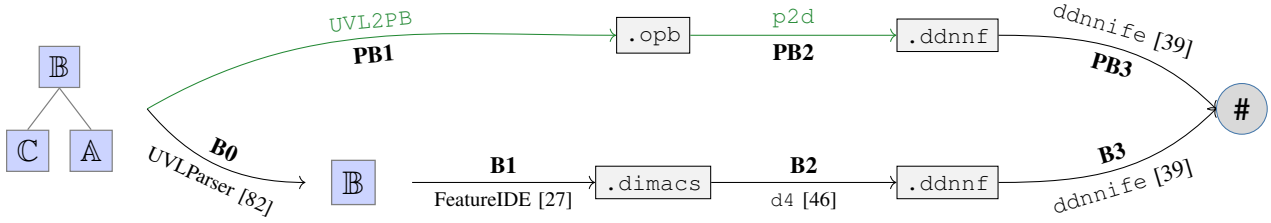


Fig. 5. Comparison of Boolean and Pseudo-Boolean Approach for Compiling Expressive Feature Models to d-DNNF

c) Subject Systems: Currently, there is a lack of industrial feature models with expressive constructs in the literature. Even though commercial tools support expressive constructs [22], [23], [29] and such constructs are used in practice [30]–[32], feature models created with these tools are not *publicly* available. With our selection of subject systems, we aim to provide indicators on the performance of our approach in practice by evaluating on real-world basic feature models and synthesized feature models with expressive constructs. Table VI gives an overview on the different datasets. Each evaluated feature model is given in UVL format.

TABLE VI
OVERVIEW SUBJECT SYSTEMS

Dataset	#Models	#Variables	Constructs
Literature Collection	76	100–80,258	$\mathbb{B}$
Confidential Industry	2	612–645	$\mathbb{B}$ & $\mathbb{A}$
Single Construct	16,164	1–8,000	Isolated from $\mathbb{C}$ & $\mathbb{A}$
Random	2,700	80–2,767	Combs. from $\mathbb{B}$, $\mathbb{C}$, & $\mathbb{A}$

Real-World Basic Feature Models Most publicly available feature models are limited to basic constructs [8], [11], [12], [17]. In addition, we expect that even feature models with more extended expressions contain a considerable share of basic constraints. Hence, we evaluate our approach on real-world basic feature models. We use a dataset recently collected from industrial models in the literature [12]. The full provided dataset contains several histories of feature models with hundreds of similar feature models. For every history, we use three feature models with the minimum, median, and maximum number of constraints respectively instead of the entire history. A configuration file specifying which models have been used can be found here.[10] The used dataset covers various domains (e.g., automotive and systems software) and varying sizes (100–80,258 variables and 0–388,816 constraints).

Industry Models with Attribute Constraints. Unfortunately, there is a lack of real-world feature models with expressive constructs in the literature. From our industry partner, we have access to two feature models that each contain numerous numeric constraints over feature attributes. Overall, the feature models include constructs from basic feature models $\mathbb{B}$ and attribute constraints $\mathbb{A}$, but no cardinalities ($\mathbb{C}$).

Single Construct Models. With the third dataset, we examine the performance of our pseudo-Boolean pipeline for different language constructs in isolation. In particular, we generate feature models including only one of alternative, feature cardinality, group cardinality, and numeric constraints and if needed optional features. Table VII provides an overview on the generated models for the specific constructs. For each incarnation of a specific construct, we create a feature model only including features and constraints corresponding to this incarnation. Depending on the construct, we create different combinations of relevant parameters to achieve a wide coverage. As an example, for alternatives, we create 80 feature models consisting of one alternative with between 100 and 8,000 features each. To evaluate feature cardinality, we create a single feature with that cardinality and attach four groups, each having one type of the *basic* hierarchical groups and three child features. Here, we vary the number of features in the group, the lower bound of the cardinality, and the upper bound. For the numeric constraints, we evaluate constraints consisting of growing expressions A of the respective operators and a constant $\mathbb{E}_p(A)$ for comparison. The constant $\mathbb{E}_p(A)$ controls that a share p of the features need to be selected on average to satisfy the constraint. So for $p = 0.1$, the value $\mathbb{E}_p(A)$ is set to a value that is met or exceeded when selecting 10 % of the features on average.

Generated UVL Models. To examine the performance of the different approaches, when dealing with different combinations of the constructs, we randomly generate further UVL models using the UVLGenerator [100].[11] To this end, We aim to cover a wide range of structural properties and distribution of constructs. We generate feature models that include basic constructs and additional expressive constructs by following three strategies: (1) *exactly one* expressive construct, (2) *all but one* expressive construct, (3) all expressive constructs. For each set of included expressive construct, we create 300 feature models which are separated in three size clusters: small, medium, and large. The size clusters are derived from statistics over the collection of real-world feature models [12]. We use the minimal[12] (100), median (554), and third quartile (2,306) number of features from the collection as point of reference for the small, medium, and large models. The generated feature models have the respective reference value +/- 20 % number of features. We configured other structural properties (e.g., tree depth and distribution of group types) in the configurator to match the corresponding distributions of that property in the

[10]https://github.com/SoftVarE-Group/feature-model-benchmark/blob/master/pre_configs/minmedmax.json

[11]https://github.com/SoftVarE-Group/uvlgenerator

[12]In the collection [12], only feature models with 100 or more features are considered.

TABLE VII
OVERVIEW SINGLE CONSTRUCT MODELS

Construct	How?
Alternative	$alt^{p}_{c_1,..c_n}$ with $n \in \{100, 200, \ldots, 8\,000\}$
Group cardinality	$[x..y]^{p}_{c_1,..,c_n}$ with $\forall x, y : x < y$
Feature cardinality	$f[\lceil x \rceil .. \lceil y \rceil]$ with $\forall x, y : x < y$
	$x, y \in \{1, 0.25n, 0.5n, 0.75n, n\}$
	$n \in \{1, 2, \ldots, 100, 200, \ldots 2\,000\}$
Numeric constraints	$\sum_{i=1}^{n} a_i \geq b$ with $b = \mathbb{E}_p(\sum_{i=1}^{n} a_i)$
	$n \in \{1, 2, \ldots, 100, 200, \ldots 2\,000\}$
	$\prod_{i=1}^{n} a_i + 1 \geq b$ with $b = \mathbb{E}_p(\prod_{i=1}^{n} a_i + 1)$
	$n \in \{2, 3, \ldots, 18\}$
	$\frac{a_1 \star \ldots \star a_{n/2}}{a_{n/2+1} \star \ldots \star a_n} > b$ with $b = \mathbb{E}_p(\frac{a_1 \star \ldots \star a_{n/2}}{a_{n/2+1} \star \ldots \star a_n})$
	$n \in \{4, 6, \ldots, 18\}$, $p \in \{0.1, \ldots, 0.9\}$

feature-model collection [12].

d) Experiment Execution: For each of the feature models, we evaluate the whole pseudo-Boolean and Boolean pipeline (cf. Figure 5). The pseudo-Boolean pipeline consists of translating the UVL model to pseudo-Boolean formula with our encoding (referred to as PB1), compile the formula with `p2d` to d-DNNF (PB2), and compute the model counts produced by the d-DNNF with `ddnnife` (PB3). For the Boolean pipeline, we convert the UVL model to a basic feature model using UVL conversion strategies [21] (B0), translate the basic feature model to CNF with FeatureIDE [27] (B1), compile the CNF to d-DNNF with `d4` [46] (B2), and check the model count produced by the d-DNNF with `ddnnife` [39] (B3). The timeouts are ten minutes for conversion to pseudo-Boolean/CNF and ten minutes for compilation. We collect the runtimes for the pipelines, which consist of PB1 + PB2 and B0 + B1 + B2, respectively. To inspect whether a runtime difference is statistically significant, we use a Wilxocon signed-ranked test [101] and assume the significance level $\alpha = 0.05$. Furthermore, we inspect the sizes of the formula encodings from PB1 and B1 and of the compiled d-DNNFs from PB2 and B2. As indicator for the correctness of our approach, we compare the model counts produced by `ddnnife` for every model (i.e., PB3 vs B3).

B. Results

In the following, we present and discuss the results of our empirical evaluation. For every successfully analyzed feature model, the d-DNNFs compiled by `p2d` and `d4` produce the same model count. Since we evaluate on 18,942 models with combinations of constructs, this is a strong indicator that both our pseudo-Boolean encoding and `p2d` are correct.

a) Encoding (RQ1): Table VIII provides an overview on the performance of the Boolean and pseudo-Boolean pipelines. For each dataset, we compare the share of models that have been successfully encoded/compiled by each strategy and the respective sizes (median over models successfully encoded/compiled by both) of the input formula in CNF/pseudo-Boolean and the d-DNNFs. The pseudo-Boolean encoding is significantly faster for every dataset, including the basic feature models. The difference in encoding performance is especially apparent for group cardinality, where all isolated

TABLE VIII
BOOLEAN VS PSEUDO BOOLEAN: PERFORMANCE OVERVIEW

Dataset	Enc. (%)		Enc. Size		Comp. (%)		d-DNNF Size	
	CNF	PB	CNF	PB	`d4`	`p2d`	`d4`	`p2d`
Real-World Basic	97.4	97.4	5,073	**3.987**	**81.6**	80.3	**2,310**	2,892
Isolated								
Alternative	98.8	**100**	1.6M	**4,002**	12.5	**53.8**	2,751	2,751
Group Cardinality	7.05	**100**	42	**22**	7.05	**87.4**	44	44
Feature Cardinality	79.1	**99.0**	42K	**2,486**	70.7	**76.8**	26k	**1,800**
Addition	9.15	**100**	37	**26**	9.15	**90.5**	**40.5**	59.5
Multiplication	60.9	**94.7**	**32**	986	60.9	**76.2**	**39**	308
Division	48.1	**100**	**22**	135	48.1	**100**	**20**	46
Synthesized								
Just Group Card.	81.7	**100**	9,709	**8,667**	81.7	**100**	614	**613**
Just Feature Card.	97.7	**100**	41K	**21K**	96.7	96.7	1,257	**970**
Just Expression	100	100	9,860	**9,191**	100	100	**686**	795
Just Aggregate	3.67	**100**	9,411	**7,979**	3.67	**100**	489	489
All but Group Card.	2.0	**100**	5,202	**3,983**	2.0	**97.3**	151	**150**
All but Feature Card.	4.33	**100**	2,140	**1,964**	4.33	**99.3**	125	**113**
All but Expression	8.33	**99.7**	88K	**35K**	8.33	**98.3**	1,313	**872**
All but Aggregate	81.7	**100**	34K	**18K**	81.0	**97.7**	1,072	**973**
All constructs	6.0	**100**	11K	**5,384**	6.0	**96.3**	276	**186**
Industrial	0	**100**	N.A.	**14K**	0	0	N.A.	N.A.

models with up-to 5,000 features could be encoded with pseudo-Boolean, but only 7.05 % (up-to 13 features) with the Boolean encoding. Similarly, all sums (up-to 2,100 features) could be encoded with pseudo-Boolean, but only 9.15% (up-to 19 features) for Boolean. For multiplication (94.7% vs 60.9 %) and division (100 % vs 48.1 %), the differences are smaller, but still considerably advantageous for pseudo-Boolean. For the two expressive feature models from our industry partner, the pseudo-Boolean encoding required 1.44 and 1.51 seconds, while Boolean hit the timeout of ten minutes. For alternatives, group cardinalities, and feature cardinalities, pseudo-Boolean produces over 90 % fewer literals. On the other side, for multiplication and division the Boolean encoding produces 98.4 % and 83.6 % smaller formulas.

RQ1: On every evaluated dataset, including basic feature models without any expressive constructs, pseudo-Boolean encoding is significanly faster than Boolean encoding. For some constructs, pseudo-Boolean encoding scales for thousands of features while Boolean encoding hits the timeout of ten minutes already for a few dozen features. The resulting pseudo-Boolean formulas are also smaller for every dataset except multiplications and divisions in isolation.

b) Compiling Basic Feature Models (RQ2): Figure 6 shows the runtimes of Boolean and pseudo-Boolean compilation without encoding time (left side) and with encoding time (right side). Timeouts from at least one of the pipelines are marked with gray crosses $\times$. For 11 out of the 74 successfully encoded feature models, both compilers resulted in a timeout. Respectively, `p2d` failed to compile for two other models and `d4` for one other. `p2d` was faster for 32 models and `d4` for 31. `d4` required less overall runtime with 11.8 vs 15.9 minutes. However, the difference between both compilers is not statistically significant ($p = 0.06$). `d4` produces smaller d-DNNFs overall (2,310 vs 2,892 nodes in the median).

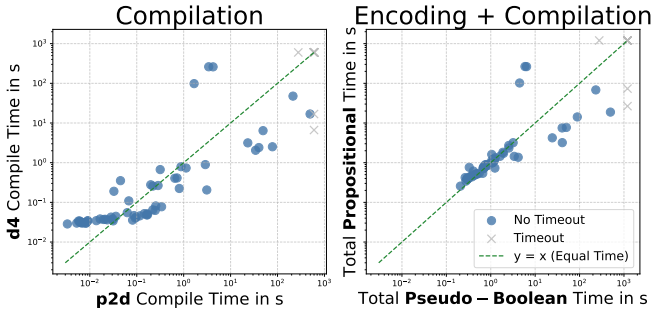


Fig. 6. Compilation Comparison: Basic Feature Models

RQ2: Our results suggest that our approach is competitive with state-of-the-art Boolean d-DNNF compilation even for basic feature models. Even though `d4` requires less runtime (but not significantly) for the dataset of basic feature models, `p2d` is faster for about half of the models (32 of 63 successfully compiled). However, `p2d` generally produces larger d-DNNFs, which may impact runtime of follow-up analyses. It may be valuable to optimize the size of compiled d-DNNFs in future work on `p2d`.

c) Compiling Expressive Constructs (RQ3): Figure 7 shows the runtimes of Boolean and pseudo-Boolean compilation including encoding times for the *constructs in isolation*. Here, the timeouts are omitted. Note that for addition and group cardinality we only show an excerpt of the instances solved by `p2d` as the performance of `d4` is hard to observe otherwise. The pseudo-Boolean approach scales for a higher number of features for all constructs. For group cardinality, `p2d` successfully compiled 87.4 % including ones from the largest category (5,000 features), while only 7.05 % could be *encoded in CNF*. Nevertheless, each feature model encoded in CNF could be compiled to d-DNNF with `d4`. For additions, the difference is also substantial as 90.5 % (up-to 2,100 features) can be compiled with pseudo-Boolean, but only 9.15 % (up-to 19 features) with the Boolean pipeline. The differences for alternative (53.8 % vs 12.5 %), feature cardinality (76.8 % vs 70.7 %), multiplication (76.2 % vs 60.9 %) and division (100 % vs 48.1 %) are small but also considerable.

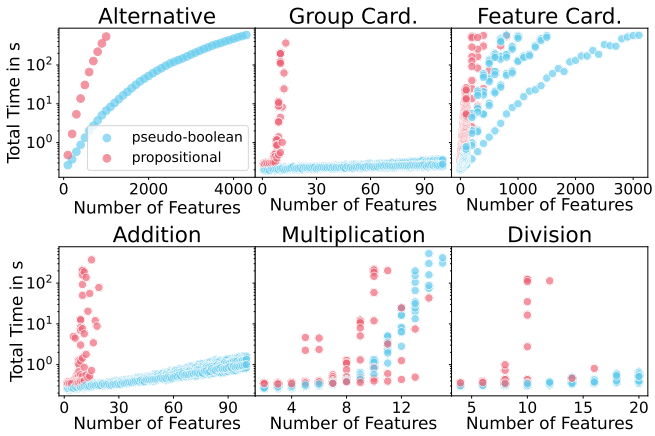


Fig. 7. Compilation Comparison: Isolated Models

Figure 8 shows the comparison for the different synthesized feature-model datasets. `p2d` is significantly faster for every dataset except *just expressions* (i.e., feature model with basic constructs and numeric expressions over attributes) where it still requires less runtime overall. Over the 2,700 analyzed feature models, the pseudo-Boolean pipeline successfully evaluated 2,657 (98.4 %) and the Boolean pipeline 1,156 (42.6 %). For both real-world models from our industry partner, compilation with `p2d` hit the timeout of ten minutes, while for Boolean, already the encoding step failed.

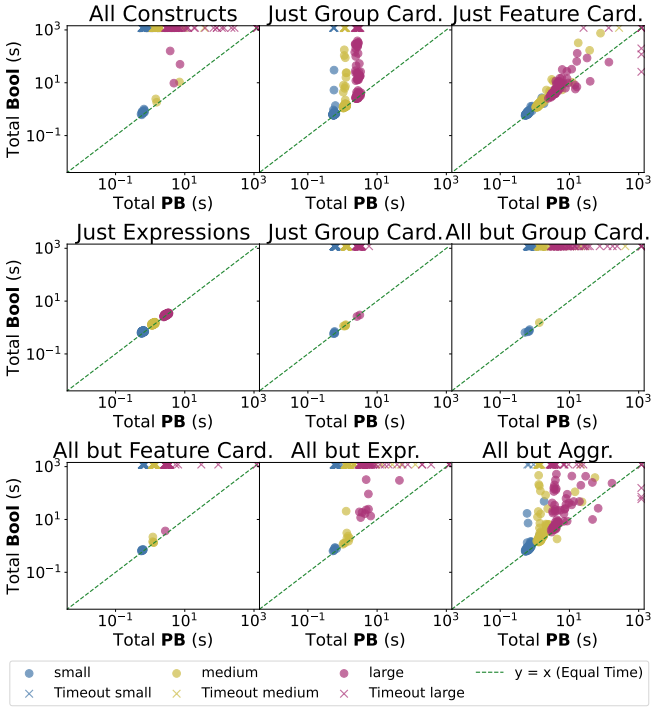


Fig. 8. Compilation Comparison: Synthesized Models

The sizes of compiled d-DNNFs are generally similar, with small advantages for `p2d` on most datasets. However, for multiplication and division the d-DNNFs compiled by `p2d` are considerably larger, probably due to the high number of artificial variables introduced by the encoding. This is especially observable in the isolated models where the d-DNNFs compile in the median to 308 (`p2d`) vs 39 (`d4`) nodes for multiplication and 46 vs 20 nodes for division.

RQ3: For expressive feature-modeling constructs, pseudo-Boolean d-DNNF compilation yields substantial runtime benefits over the Boolean state-of-the-art. Especially if feature models contain aggregate functions or group cardinalities, our pseudo-Boolean approach substantially outperforms the Boolean baseline. Still, the results on the two industrial models and the encoding size for multiplication and division suggest that further optimization on our approach may be beneficial.

C. Threats to Validity

a) Computational Bias: The performance of running one of the tools can differ for the same input instances due to

computational bias. To reduce such a bias in our conclusions, we repeated each measurement three times and used the median. The impact is further reduced by the deterministic behavior of each tool, also indicated by the little variance within the three repetitions. Paired with the high number of evaluated models per tool, we do not suppose a considerable impact of computational bias on our conclusions.

b) Conversion to Logic: Alternative strategies to translate the feature models to Boolean and pseudo-Boolean logic may impact the performance [60]. For Boolean logic, a plethora of different encodings for constructs such as at-least-k and at-most-k [24], or numerical variables [51] have been considered. However, their suitability (w.r.t. semantic equivalence) or performance is unknown for encoding feature models and d-DNNF compilation. We discuss this further in Section VI. For pseudo-Boolean logic, we are not aware of alternative strategies in the literature to translate the constructs, but there should be numerous semantically equivalent encodings for the constructs. However, adding more encodings for Boolean or pseudo-Boolean would add another layer of complexity to the already complex evaluation. While out of scope for this work, considering further strategies may be valuable in the future for both Boolean and pseudo-Boolean encoding.

c) Tool Parameters: Both compilers, `d4` [46] and `p2d`, can be parametrized, which can have an impact on the performance. However, considering the combinatorics of different parameters would also vastly increase the complexity of the evaluation and, thus, we consider it out of scope. For the evaluation, we used the default parameters for every considered tool.

d) Transferability of Industrial Models: Our selection of real-world models may not be transferrable to other feature models. However, the collection used is the result of a broad literature survey to identify available real-world models in the literature [12]. The set covers various domains and properties, such as number of features, number of constraints, or tree structure.

e) Transferability of Synthesized Models: The performance results on the synthesized models may not be transferable to practice. However, we are not aware of industrial feature models with the expressive constructs available in the literature. With our synthesized dataset, we aim to showcase the performance for different instances in isolation and also in combination with basic and other expressive constructs. Furthermore, for the synthesized feature models, we considered structural properties of industrial feature models for the generation. Nevertheless, we still consider an evaluation of our approach on more real-world models with expressive constructs as valuable future work.

VI. Related Work

a) Pseudo-Boolean Solving: Some pseudo-Boolean solvers have been proposed [54], [78], [102]. However, these are limited to single-invocation satisfiability checks. Morgado et al. [102] use a pseudo-Boolean SAT solver to count by employing *blocking clauses*. Here, a SAT solver is iteratively queried and returned solutions are negated and added to the formula as blocking clauses. Querying the solver on the updated formula again then always provides a different solution. This procedure is then repeated til all solutions are *blocked* to find the overall number of satisfying assignments. The approach of using blocking clauses however requires SAT calls up-to the number of satisfying assignments on an ever growing formula and is generally seen as not scalable for formulas with many satisfying assignments [103]. Feature models typically induce a very high number of solutions, often with tens, hundreds, or even thousands of digits [12], [97], [104] (e.g., Linux induces more than 10^{616} solutions). Very recently another pseudo-Boolean knowledge compiler has been proposed by Yang and Meel [55]. Their approach compiles to algebraic decision diagrams which are not limited to Boolean variables and generally more complex to compile. In preliminary experiments, we compared their compiler to `p2d`, but their compiler often produced inconsistent (compared to the results of `p2d` and `d4`) model counts and is likely unsound. Hence, we did not consider their compiler in our empirical evaluation. Nevertheless, the preliminary experiments also showed that `p2d` requires substantially less runtime for compiling the feature models.

b) d-DNNFs in Feature-Model Analysis: d-DNNFs enable linear time (w.r.t. number of nodes in the d-DNNF) queries for satisfiability, counting, and enumeration [13], [44]. A plethora of feature-model analyses rely on potentially numerous of these operations [13] and, thus, can benefit from d-DNNFs. In related literature d-DNNFs have already been employed for counting [39], [56], sampling [40], [56], satisfiability checks [56], and deriving configurations optimized for different objectives [56].

c) Boolean d-DNNF Compilation: Three Boolean d-DNNF compilers are commonly considered in the literature, namely `c2d` [42], `dSharp` [45], and `d4` [46]. For our empirical evaluation, we only considered `d4` as baseline because the compiler substantially outperformed `c2d` and `dSharp` in previous evaluations on feature models [13], [97]. Our compiler `p2d` adopts many of the strategies employed in the three Boolean compilers. Still, the type of input formula is vital for the realization of a compiler, as internal datastructures and optimizations heavily rely on the exact structure (i.e., CNF for Boolean compilers). Thus, it is not trivial to adapt an existing Boolean d-DNNF compiler to consider pseudo-Boolean formulas.

d) Expressive Feature-Model Dialects: While reasoning for feature-model analysis mostly focuses on Boolean logic in CNF, more expressive constructs (e.g., group cardinality) were considered in a large variety of work. ter Beek et al. [105] provide an overview on textual formats for specifying feature models. 11 out of 13 identified formats support at least one of the expressive constructs facilitated by pseudo-Boolean encoding. Horcas et al. [106] suggest a metamodel approach to cover constructs from different variability languages. Their proposal also includes various constructs, such as arithmetic expressions and cardinalities. In the initial publication of UVL, results of a community questionnaire imply that expressive constructs such as cardinality groups and feature attributes are desired [80]. Various variability-modeling tools that are com-

monly used in practice [25], such as `pure::variants` [23], DOPLER [29], and Gears [22] also supports more expressive constraints. Furthermore, several case studies on industrial systems imply that the expressive constructs we targeted are mandated [20], [24], [30]–[32]. In summary, insights from related work suggest that more expressive constructs are of interest for practitioners and researchers. The plethora of variability languages, including UVL, can benefit from our advances in scalability for feature-model analysis on these expressive constructs.

e) Encodings for Expressive Feature-Modeling Constructs: While knowledge compilation for expressive feature-modeling constructs has not been yet employed, there has been some works on *encoding* these constructs. Various publications present for cardinality constraints [24], [83], [84] which can be used to model cardinalities. However, many of these encodings only preserve equisatisfiability [24], [83], [84], which can produce faulty results for model counting and for subsequent analyses on a compiled d-DNNF. Furthermore, the performance of these encodings when applied to feature models in general and as input for knowledge compilation is unknown. Nevertheless, examining the plethora of available encodings may reveal suitable and possibly more efficient candidates. We consider the analysis of existing encodings as valuable future work, but as out of scope for this work as it substantially increases the complexity of our evaluation. Benavides et al. [36] propose to encode extended (i.e., attributed) feature models with constraint programming. However, their approach only scales to very small feature models (i.e., fewer than 25 features), while our strategy allows analyzing feature models with thousands of features. Karataş et al. [68] also encode extended feature models with cardinality groups with constraint programming. However, their approach also employs one-shot computations instead of knowledge compilation. Munoz et al. [51] employ bit-blasting to represent numeric features and constraints over them in Boolean logic. However, they did not consider other expressive constructs such as cardinalities. While we did not consider numeric features in this work, the bit-blasting strategy could also be valuable to apply for pseudo-Boolean logic.

f) Reasoning for Expressive Formulas: Satisfiability Modulo Theories (SMT) supports various expressive constructs, such as typed features and constraints over them [33], [65], [107]. SMT solvers are available for satisfiability checking [33], [34] and approximate counting [108]. Constraint programming has been also applied for feature-model analysis [36], [37] to represent basic constructs [36], [37] and attribute constraints [36]. However, for neither of these formats, exact model counters or knowledge compilers are available to the best of our knowledge. Furthermore, the increased expressive power of SMT and CP comes with added computational complexity [36], [37], [109]. With our approach of pseudo-Boolean d-DNNF compilation, we aim to hit a sweet-spot between more complex reasoning on expressive formats and the state-of-the-art, which relies on the rather restrictive format CNF [15], [27], [66], [67].

VII. Conclusion

There is a mismatch between the expressiveness of available variability languages [9] and available reasoning engines [15], [97]. Most variability languages allow various expressions [9] that are not well suited for Boolean encoding. However, state-of-the-art feature-model analysis is mainly focused on Boolean logic [13], [15], [27], [51], [66], [67], [99]. Matching expectations and previous observations [24], our evaluation provides more evidence that encoding common constructs, such as cardinalities or attribute constraints, in Boolean logic comes with substantial scalability issues.

In our work, we tackle this issue by introducing pseudo-Boolean d-DNNF compilation. Our approach consists of a pseudo-Boolean encoding for feature models and a pseudo-Boolean d-DNNF compiler. As the compiled d-DNNF is Boolean, we can employ existing algorithms and tools that enable efficient feature-model analyses [39], [40], [56]. Our empirical evaluation clearly suggests that using our pseudo-Boolean approach yields substantial runtime benefits over the Boolean state-of-the-art. We did not tackle all feature-modeling constructs, as, for instance, we are not able to naturally represent features with other types (e.g., numeric) with pseudo-Boolean logic. Still, pseudo-Boolean d-DNNF compilation considerably advances what feature-modeling constructs can be efficiently analyzed. We envision that the availability of scalable reasoning increases the usage of these constructs, as automated analysis is vital for product-line engineering [2], [3], [14], [98], [110]–[112].

Novelty Statement

Our work reports completely new research results and is not an extension of any existing work. None of our main contributions, namely the pseudo-Boolean encoding, pseudo-Boolean d-DNNF compilation, the tooling, and the empirical evaluation, have been or will be presented or published in other work.

References

[1] D. Eichhorn, T. Pett, N. Przigoda, J. Kindsvater, C. Seidl, and I. Schaefer, "Coverage-driven test automation for highly-configurable railway systems," in *Proc. Int'l Working Conf. on Variability Modelling of Software-Intensive Systems (VaMoS)*. New York, NY, USA: ACM, 2023, p. 23–30.

[2] A. Kübler, C. Zengler, and W. Küchlin, "Model Counting in Product Configuration," in *Proc. Int'l Workshop on Logics for Component Configuration (LoCoCo)*. Open Publishing Association, 2010, pp. 44–53.

[3] O. Oliinyk, K. Petersen, M. Schoelzke, M. Becker, and S. Schneickert, "Structuring Automotive Product lines and Feature Models: An Exploratory Study at Opel," *Requirements Engineering*, vol. 22, no. 1, pp. 105–135, Mar. 2017.

[4] T. F. Bowen, F. S. Dworack, C.-H. Chow, N. Griffeth, G. E. Herman, and Y.-J. Lin, "The Feature Interaction Problem in Telecommunications Systems," in *Proc. Int'l Conf. on Software Engineering for Telecommunication Switching Systems (SETSS)*. IEEE, 1989, pp. 59–62.

[5] E. Kowalczyk, M. B. Cohen, and A. M. Memon, "Configurations in Android Testing: They Matter," in *Proc. Int'l Workshop on Advances in Mobile App Analysis (A-Mobile)*. ACM, 2018, pp. 1–6.

[6] R. Lotufo, S. She, T. Berger, K. Czarnecki, and A. Wąsowski, "Evolution of the Linux Kernel Variability Model," in *Proc. Int'l Systems and Software Product Line Conf. (SPLC)*. Springer, 2010, pp. 136–150.

[7] D. Romero-Organvidez, P. Neira, J. A. Galindo, and D. Benavides, "Kconfig Metamodel: A First Approach," in *Proc. Int'l Systems and Software Product Line Conf. (SPLC)*. ACM, 2024, pp. 55–60.

[8] T. Berger, S. She, R. Lotufo, A. Wąsowski, and K. Czarnecki, "A Study of Variability Models and Languages in the Systems Software Domain," *IEEE Trans. on Software Engineering (TSE)*, vol. 39, no. 12, pp. 1611–1640, 2013.

[9] M. H. t. Beek, K. Schmid, and H. Eichelberger, "Textual Variability Modeling Languages: An Overview and Considerations," in *Proc. Int'l Workshop on Languages for Modelling Variability (MODEVAR)*. ACM, 2019, pp. 151–157.

[10] D. Batory, "Feature Models, Grammars, and Propositional Formulas," in *Proc. Int'l Systems and Software Product Line Conf. (SPLC)*. Springer, 2005, pp. 7–20.

[11] A. Knüppel, T. Thüm, S. Mennicke, J. Meinicke, and I. Schaefer, "Is There a Mismatch Between Real-World Feature Models and Product-Line Research?" in *Proc. Europ. Software Engineering Conf./Foundations of Software Engineering (ESEC/FSE)*. ACM, 2017, pp. 291–302.

[12] C. Sundermann, V. F. Brancaccio, E. Kuiter, S. Krieter, T. Heß, and T. Thüm, "Collecting Feature Models from the Literature: A Comprehensive Dataset for Benchmarking," in *Proc. Int'l Systems and Software Product Line Conf. (SPLC)*. ACM, 2024, pp. 54–65.

[13] C. Sundermann, E. Kuiter, T. Heß, H. Raab, S. Krieter, and T. Thüm, "On the Benefits of Knowledge Compilation for Feature-Model Analyses," *Annals of Mathematics and Artificial Intelligence (AMAI)*, vol. 92, no. 5, pp. 1013–1050, 2024.

[14] D. Benavides, S. Segura, and A. Ruiz-Cortés, "Automated Analysis of Feature Models 20 Years Later: A Literature Review," *Information Systems*, vol. 35, no. 6, pp. 615–708, 2010.

[15] J. H. Liang, V. Ganesh, K. Czarnecki, and V. Raman, "SAT-Based Analysis of Large Real-World Feature Models Is Easy," in *Proc. Int'l Systems and Software Product Line Conf. (SPLC)*. Springer, 2015, pp. 91–100.

[16] C. Sundermann, M. Nieke, P. M. Bittner, T. Heß, T. Thüm, and I. Schaefer, "Applications of #SAT Solvers on Feature Models," in *Proc. Int'l Working Conf. on Variability Modelling of Software-Intensive Systems (VaMoS)*. ACM, 2021.

[17] J. Oh, P. Gazzillo, D. Batory, M. Heule, and M. Myers, "Uniform Sampling From Kconfig Feature Models," University of Texas at Austin, Department of Computer Science, Tech. Rep. TR-19-02, 2019.

[18] C. Sundermann, E. Kuiter, T. Heß, H. Raab, S. Krieter, and T. Thüm, "On the Benefits of Knowledge Compilation for Feature-Model Analyses," in *Proc. Int'l Systems and Software Product Line Conf. (SPLC)*. ACM, 2024, p. 217.

[19] K. Feichtinger, C. Sundermann, T. Thüm, and R. Rabiser, "It's Your Loss: Classifying Information Loss During Variability Model Roundtrip Transformations," in *Proc. Int'l Systems and Software Product Line Conf. (SPLC)*. ACM, 2022, pp. 67–78.

[20] L. Passos, M. Novakovic, Y. Xiong, T. Berger, K. Czarnecki, and A. Wąsowski, "A Study of Non-Boolean Constraints in Variability Models of an Embedded Operating System," in *Proc. Int'l Workshop on Feature-Oriented Software Development (FOSD)*, ser. Proc. Int'l Systems and Software Product Line Conf. (SPLC). ACM, 2011.

[21] D. Benavides, C. Sundermann, K. Feichtinger, J. A. Galindo, R. Rabiser, and T. Thüm, "Uvl: Feature modelling with the universal variability language," *J. Systems and Software (JSS)*, vol. 225, 2025.

[22] Big Lever Software Inc., "Gears: A Software Product Line Engineering Tool," Website: http://www.biglever.com/solution/product.html, 2010, accessed: 2010-11-04.

[23] D. Romano, K. Feichtinger, D. Beuche, U. Ryssel, and R. Rabiser, "Bridging the Gap Between Academia and Industry: Transforming the Universal Variability Language to Pure::Variants and Back," in *Proc. Int'l Workshop on Languages for Modelling Variability (MODEVAR)*, ser. Proc. Int'l Systems and Software Product Line Conf. (SPLC). ACM, 2022, pp. 123–131.

[24] P. M. Bittner, T. Thüm, and I. Schaefer, "SAT Encodings of the At-Most-k Constraint – A Case Study on Configuring University Courses," in *Proc. Int'l Conf. on Software Engineering and Formal Methods (SEFM)*, P. C. Ölveczky and G. Salaün, Eds. Springer, 2019, pp. 127–144.

[25] T. Berger, R. Rublack, D. Nair, J. M. Atlee, M. Becker, K. Czarnecki, and A. Wąsowski, "A Survey of Variability Modeling in Industrial Practice," in *Proc. Int'l Workshop on Variability Modelling of Software-Intensive Systems (VaMoS)*. ACM, 2013, pp. 7:1–7:8.

[26] pure::systems, "pure::variants," Website: http://www.pure-systems.com/products/pure-variants-9.html, 2017, accessed: 2017-05-10.

[27] J. Meinicke, T. Thüm, R. Schröter, F. Benduhn, T. Leich, and G. Saake, *Mastering Software Variability With FeatureIDE*. Springer, 2017.

[28] C. Sundermann, T. Heß, D. Engelhardt, R. Arens, J. Herschel, K. Jedelhauser, B. Jutz, S. Krieter, and I. Schaefer, "Integration of UVL in FeatureIDE," in *Proc. Int'l Workshop on Languages for Modelling Variability (MODEVAR)*. ACM, 2021, pp. 73–79.

[29] D. Dhungana, P. Grünbacher, and R. Rabiser, "The DOPLER Meta-Tool for Decision-Oriented Variability Modeling: A Multiple Case Study," *Automated Software Engineering*, vol. 18, no. 1, pp. 77–114, 2011.

[30] A. Hubaux, Q. Boucher, H. Hartmann, R. Michel, and P. Heymans, "Evaluating a textual feature modelling language: Four industrial case studies," in *Proc. Int'l Conf. on Software Language Engineering (SLE)*. Berlin, Heidelberg: Springer, 2011, pp. 337–356.

[31] T. Berger, D. Nair, R. Rublack, J. M. Atlee, K. Czarnecki, and A. Wąsowski, "Three Cases of Feature-Based Variability Modeling in Industry," in *Proc. Int'l Conf. on Model Driven Engineering Languages and Systems (MODELS)*. Springer, 2014, pp. 302–319.

[32] S. Wang, S. Ali, T. Yue, and M. Liaaen, "Using feature model to support model-based testing of product lines: An industrial case study," in *Proc. Int'l Conf. on Quality Software (QSIC)*, 2013, pp. 75–84.

[33] L. De Moura and N. Bjørner, "Z3: An Efficient SMT Solver," in *Proc. Int'l Conf. on Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*. Springer, 2008, pp. 337–340.

[34] J. Sprey, C. Sundermann, S. Krieter, M. Nieke, J. Mauro, T. Thüm, and I. Schaefer, "SMT-Based Variability Analyses in FeatureIDE," in *Proc. Int'l Working Conf. on Variability Modelling of Software-Intensive Systems (VaMoS)*. ACM, 2020.

[35] N. Jussien, G. Rochart, and X. Lorca, "Choco: An Open Source Java Constraint Programming Library," in *Proc. Workshop on Open-Source Software for Integer and Contraint Programming (OSSICP)*. CCSD-HAL, 2008.

[36] D. Benavides, P. Trinidad, and A. Ruiz-Cortés, "Using Constraint Programming to Reason on Feature Models," in *Proc. Int'l Conf. on Software Engineering and Knowledge Engineering (SEKE)*, 2005, pp. 677–682.

[37] R. Pohl, K. Lauenroth, and K. Pohl, "A Performance Comparison of Contemporary Algorithmic Approaches for Automated Analysis Operations on Feature Models," in *Proc. Int'l Conf. on Automated Software Engineering (ASE)*. IEEE, 2011, pp. 313–322.

[38] J. A. Galindo, M. Acher, J. M. Tirado, C. Vidal, B. Baudry, and D. Benavides, "Exploiting the Enumeration of All Feature Model Configurations: A New Perspective With Distributed Computing," in *Proc. Int'l Systems and Software Product Line Conf. (SPLC)*. ACM, 2016, pp. 74–78.

[39] C. Sundermann, H. Raab, T. Heß, T. Thüm, and I. Schaefer, "Reusing d-DNNFs for Efficient Feature-Model Counting," *Trans. on Software Engineering and Methodology (TOSEM)*, vol. 33, no. 8, 2024.

[40] S. Sharma, R. Gupta, S. Roy, and K. S. Meel, "Knowledge Compilation Meets Uniform Sampling," in *Proc. Int'l Conf. on Logic for Programming, Artificial Intelligence, and Reasoning*. EasyChair, 2018, pp. 620–636.

[41] R. Heradio, H. J. Pérez-Morago, D. Fernández-Amorós, R. Bean, F. J. Cabrerizo, C. Cerrada, and E. Herrera-Viedma, "Binary Decision Diagram Algorithms to Perform Hard Analysis Operations on Variability Models," in *Proc. Int'l Conf. on Intelligent Software Methodologies, Tools and Techniques (SOMET)*. IOS Press, 2016, pp. 139–154.

[42] A. Darwiche, "A Compiler for Deterministic, Decomposable Negation Normal Form," in *Proc. Conf. on Artificial Intelligence (AAAI)*. AAAI Press, 2002, pp. 627–634.

[43] R. E. Bryant, "Binary Decision Diagrams," in *Handbook of Model Checking*. Springer, 2018, pp. 191–217.

[44] A. Darwiche and P. Marquis, "A Knowledge Compilation Map," *J. Artificial Intelligence Research (JAIR)*, vol. 17, no. 1, pp. 229–264, 2002.

[45] C. Muise, S. A. McIlraith, J. C. Beck, and E. I. Hsu, "Dsharp: Fast d-DNNF Compilation with sharpSAT," in *Advances in Artificial Intelligence*, L. Kosseim and D. Inkpen, Eds. Springer, 2012, pp. 356–361.

[46] J.-M. Lagniez and P. Marquis, "An Improved Decision-DNNF Compiler," in *Proc. Int'l Joint Conf. on Artificial Intelligence (IJCAI)*. International Joint Conferences on Artificial Intelligence, 2017, pp. 667–673.

[47] C. Dubslaff, N. Husung, and N. Käfer, "Configuring BDD Compilation Techniques for Feature Models," in *Proc. Int'l Systems and Software Product Line Conf. (SPLC)*. ACM, 2024, pp. 209–216.

[48] J. Loth, C. Sundermann, T. Schrull, T. Brugger, F. Rieg, and T. Thüm, "UVLS: A Language Server Protocol for UVL," in *Proc. Int'l Systems and Software Product Line Conf. (SPLC)*. ACM, 2023, pp. 43–46.

[49] D. Romero-Organvídez, J. A. Galindo, C. Sundermann, J.-M. Horcas, and D. Benavides, "UVLHub: A Feature Model Data Repository Using UVL and Open Science Principles," *J. Systems and Software (JSS)*, 2024, to appear.

[50] K. Feichtinger, J. Stöbich, D. Romano, and R. Rabiser, "TRAVART: An Approach for Transforming Variability Models," in *Proc. Int'l Working Conf. on Variability Modelling of Software-Intensive Systems (VaMoS)*. ACM, 2021, pp. 8:1–8:10.

[51] D.-J. Munoz, M. Pinto, L. Fuentes, and D. Batory, "Transforming Numerical Feature Models into Propositional Formulas and the Universal Variability Language," *J. Systems and Software (JSS)*, vol. 204, 2023.

[52] H. S. Fadhlillah, K. Feichtinger, P. Bauer, E. Kutsia, and R. Rabiser, "V4rdiac: Tooling for Multidisciplinary Delta-Oriented Variability Management in Cyber-Physical Production Systems," in *Proc. Int'l Workshop on Languages for Modelling Variability (MODEVAR)*, ser. Proc. Int'l Systems and Software Product Line Conf. (SPLC). ACM, 2022, pp. 34–37.

[53] M. Thurley, "sharpSAT - Counting Models With Advanced Component Caching and Implicit BCP," in *Proc. Int'l Conf. on Theory and Applications of Satisfiability Testing (SAT)*. Springer, 2006, pp. 424–429.

[54] D. Chai and A. Kuehlmann, "A Fast Pseudo-Boolean Constraint Solver," *IEEE Trans. on Computer-Aided Design of Integrated Circuits and Systems*, vol. 24, no. 3, pp. 305–317, 2005.

[55] S. Yang and K. S. Meel, "Engineering an Exact Pseudo-Boolean Model Counter," *Proc. Conf. on Artificial Intelligence (AAAI)*, vol. 38, no. 8, pp. 8200–8208, 2024.

[56] P. Bourhis, L. Duchien, J. Dusart, E. Lonca, P. Marquis, and C. Quinton, "Reasoning on Feature Models: Compilation-Based vs. Direct Approaches," Cornell University Library, Tech. Rep., 2023.

[57] K. C. Kang, S. G. Cohen, J. A. Hess, W. E. Novak, and A. S. Peterson, "Feature-Oriented Domain Analysis (FODA) Feasibility Study," Software Engineering Institute, Tech. Rep. CMU/SEI-90-TR-21, 1990.

[58] N. Siegmund, S. Sobernig, and S. Apel, "Attributed Variability Models: Outside the Comfort Zone," in *Proc. Europ. Software Engineering Conf./Foundations of Software Engineering (ESEC/FSE)*. ACM, 2017, pp. 268–278.

[59] M. Cordy, P.-Y. Schobbens, P. Heymans, and A. Legay, "Beyond Boolean Product-Line Model Checking: Dealing With Feature Attributes and Multi-Features," in *Proc. Int'l Conf. on Software Engineering (ICSE)*. IEEE, 2013, pp. 472–481.

[60] E. Kuiter, S. Krieter, C. Sundermann, T. Thüm, and G. Saake, "Tseitin or not Tseitin? The Impact of CNF Transformations on Feature-Model Analyses," in *Proc. Int'l Conf. on Automated Software Engineering (ASE)*. ACM, 2022, pp. 110:1–110:13.

[61] A. Biere, "PicoSAT Essentials," *J. Satisfiability, Boolean Modeling and Computation*, vol. 4, pp. 75–97, 2008.

[62] D. Le Berre and A. Parrain, "The Sat4j Library, Release 2.2," *J. Satisfiability, Boolean Modeling and Computation*, vol. 7, no. 2-3, pp. 59–64, 2010.

[63] S. Sharma, S. Roy, M. Soos, and K. S. Meel, "GANAK: A Scalable Probabilistic Exact Model Counter," in *Proc. Int'l Joint Conf. on Artificial Intelligence (IJCAI)*, vol. 19. AAAI Press, 2019, pp. 1169–1176.

[64] J. Burchard, T. Schubert, and B. Becker, "Laissez-Faire Caching for Parallel #SAT Solving," in *Proc. Int'l Conf. on Theory and Applications of Satisfiability Testing (SAT)*. Springer, 2015, pp. 46–61.

[65] L. De Moura and N. Bjørner, "Satisfiability Modulo Theories: Introduction and Applications," *Comm. ACM*, vol. 54, no. 9, pp. 69–77, 2011.

[66] J. A. Galindo, J. M. Horcas, A. Felfernig, D. Fernández-Amorós, and D. Benavides, "FLAMA: A Collaborative Effort to Build a New Framework for the Automated Analysis of Feature Models," in *Proc. Int'l Systems and Software Product Line Conf. (SPLC)*. ACM, 2023, pp. 16–19.

[67] M. Acher, P. Collet, P. Lahire, and R. B. France, "Familiar: A Domain-Specific Language for Large Scale Management of Feature Models," *Science of Computer Programming (SCP)*, vol. 78, no. 6, pp. 657–681, 2013.

[68] A. S. Karataş, H. Oğuztüzün, and A. Doğru, "From extended feature models to constraint logic programming," *Science of Computer Programming (SCP)*, vol. 78, no. 12, pp. 2295–2312, 2013.

[69] G. Bécan, R. Behjati, A. Gotlieb, and M. Acher, "Synthesis of attributed feature models from product descriptions," in *Proc. Int'l Systems and Software Product Line Conf. (SPLC)*. New York, NY, USA: ACM, 2015, pp. 1–10.

[70] L. Ochoa, O. González-Rojas, and T. Thüm, "Using Decision Rules for Solving Conflicts in Extended Feature Models," in *Proc. Int'l Conf. on Software Language Engineering (SLE)*. ACM, 2015, pp. 149–160.

[71] F. Medeiros, C. Kästner, M. Ribeiro, R. Gheyi, and S. Apel, "A Comparison of 10 Sampling Algorithms for Configurable Systems," in *Proc. Int'l Conf. on Software Engineering (ICSE)*. ACM, 2016, pp. 643–654.

[72] Q. Plazar, M. Acher, G. Perrouin, X. Devroey, and M. Cordy, "Uniform Sampling of SAT Solutions for Configurable Systems: Are We There Yet?" in *Proc. Int'l Conf. on Software Testing, Verification and Validation (ICST)*. IEEE, 2019, pp. 240–251.

[73] M. Mendonça, A. Wąsowski, and K. Czarnecki, "SAT-Based Analysis of Feature Models Is Easy," in *Proc. Int'l Systems and Software Product Line Conf. (SPLC)*. Software Engineering Institute, 2009, pp. 231–240.

[74] M. Huth and M. Ryan, *Logic in Computer Science: Modelling and Reasoning About Systems*. Cambridge University Press, 2004.

[75] K. Czarnecki and A. Wąsowski, "Feature Diagrams and Logics: There and Back Again," in *Proc. Int'l Systems and Software Product Line Conf. (SPLC)*. IEEE, 2007, pp. 23–34.

[76] C. Sundermann, J. Loth, and T. Thüm, "Efficient Slicing of Feature Models via Projected d-DNNF Compilation," in *Proc. Int'l Conf. on Automated Software Engineering (ASE)*. ACM, 2024, pp. 1332–1344.

[77] P. L. Hammer and S. Rudeanu, "Pseudo-Boolean Programming," *Operations Research*, vol. 17, no. 2, pp. 233–261, 1969.

[78] H. M. Sheini and K. A. Sakallah, "Pueblo: A hybrid pseudo-boolean sat solver," *J. Satisfiability, Boolean Modeling and Computation*, vol. 2, pp. 165–189, 2006.

[79] S. Henneberg, "Next-Generation Feature Models With Pseudo-Boolean SAT Solvers," Bachelor's Thesis, University of Passau, 2011.

[80] C. Sundermann, K. Feichtinger, D. Engelhardt, R. Rabiser, and T. Thüm, "Yet Another Textual Variability Language? A Community Effort Towards a Unified Language," in *Proc. Int'l Systems and Software Product Line Conf. (SPLC)*. ACM, 2021, pp. 136–147.

[81] M. Kowal, S. Ananieva, and T. Thüm, "Explaining Anomalies in Feature Models," in *Proc. Int'l Conf. on Generative Programming: Concepts & Experiences (GPCE)*. ACM, 2016, pp. 132–143.

[82] C. Sundermann, S. Vill, T. Thüm, K. Feichtinger, P. Agarwal, R. Rabiser, J. A. Galindo, and D. Benavides, "UVLParser: Extending UVL With Language Levels and Conversion Strategies," in *Proc. Int'l Systems and Software Product Line Conf. (SPLC)*. ACM, 2023, pp. 39–42.

[83] C. Sinz, "Towards an optimal cnf encoding of boolean cardinality constraints," in *Proc. Int'l Conf. on Principles and Practice of Constraint Programming (CP)*. Berlin, Heidelberg: Springer, 2005, pp. 827–831.

[84] O. Bailleux and Y. Boufkhad, "Efficient cnf encoding of boolean cardinality constraints," in *Proc. Int'l Conf. on Principles and Practice of Constraint Programming (CP)*. Berlin, Heidelberg: Springer, 2003, pp. 108–122.

[85] W. Klieber and G. Kwon, "Efficient CNF Encoding for Selecting 1 From n Objects," in *Proc. Int'l Workshop on Constraints in Formal Verification (CFV)*. Springer, 2007.

[86] K. Czarnecki and C. H. P. Kim, "Cardinality-Based Feature Modeling and Constraints: A Progress Report," in *Proc. Int'l Workshop on Software Factories (SF)*, 2005, pp. 16–20.

[87] L. Güthing, M. Weiß, I. Schaefer, and M. Lochau, "Sampling Cardinality-Based Feature Models," in *Proc. Int'l Working Conf. on Variability Modelling of Software-Intensive Systems (VaMoS)*. New York, NY, USA: ACM, 2024, pp. 46–55.

[88] M. W. Moskewicz, C. F. Madigan, Y. Zhao, L. Zhang, and S. Malik, "Chaff: Engineering an Efficient SAT Solver," in *Proc. Anual Conf. on Design Automation (DAC)*. ACM, 2001, pp. 530–535.

[89] S. Krieter, T. Thüm, S. Schulze, G. Saake, and T. Leich, "YASA: Yet Another Sampling Algorithm," in *Proc. Int'l Working Conf. on Variability Modelling of Software-Intensive Systems (VaMoS)*. ACM, 2020.

[90] E. Birnbaum and E. L. Lozinskii, "The Good Old Davis-Putnam Procedure Helps Counting Models," *J. Artificial Intelligence Research (JAIR)*, vol. 10, pp. 457–477, 1999.

[91] T. Korhonen and M. Järvisalo, "SharpSAT-TD in Model Counting Competitions 2021-2023," Cornell University Library, Tech. Rep. arXiv:2308.15819, 2023.

[92] N. Froleyks, M. Heule, M. Iser, M. Järvisalo, and M. Suda, "SAT Competition 2020," *Artificial Intelligence (AIJ)*, vol. 301, p. 103572, 2021.

[93] J. K. Fichte, M. Hecher, and F. Hamiti, "The Model Counting Competition 2020," *ACM J. of Experimental Algorithmics (JEA)*, vol. 26, 2021.
[94] T. Sang, F. Bacchus, P. Beame, H. A. Kautz, and T. Pitassi, "Combining Component Caching and Clause Learning for Effective Model Counting," in *Proc. Int'l Conf. on Theory and Applications of Satisfiability Testing (SAT)*. Springer, 2004, pp. 20–28.
[95] J. Marques-Silva, I. Lynce, and S. Malik, "Conflict-Driven Clause Learning SAT Solvers," in *Handbook of Satisfiability*. IOS Press, 2009, pp. 131–153.
[96] T. Sang, P. Beame, and H. Kautz, "Heuristics for Fast Exact Model Counting," in *Proc. Int'l Conf. on Theory and Applications of Satisfiability Testing (SAT)*. Springer, 2005, pp. 226–240.
[97] C. Sundermann, T. Heß, M. Nieke, P. M. Bittner, J. M. Young, T. Thüm, and I. Schaefer, "Evaluating State-of-the-Art #SAT Solvers on Industrial Configuration Spaces," *Empirical Software Engineering (EMSE)*, vol. 28, no. 2, p. 38, 2023.
[98] M. Hentze, C. Sundermann, T. Thüm, and I. Schaefer, "Quantifying the Variability Mismatch Between Problem and Solution Space," in *Proc. Int'l Conf. on Model Driven Engineering Languages and Systems (MODELS)*. IEEE, 2022, pp. 322–333.
[99] M. Mendonça, M. Branco, and D. Cowan, "S.P.L.O.T.: Software Product Lines Online Tools," in *Proc. Conf. on Object-Oriented Programming, Systems, Languages and Applications (OOPSLA)*. ACM, 2009, pp. 761–762.
[100] C. Sundermann, T. Heß, R. Sundermann, E. Kuiter, S. Krieter, and T. Thüm, "Generating Feature Models with UVL's Full Expressiveness," in *Proc. Int'l Workshop on Languages for Modelling Variability (MODEVAR)*. ACM, 2024, pp. 61–65.
[101] R. F. Woolson, *Wilcoxon Signed-Rank Test*. John Wiley & Sons, 2008.
[102] A. Morgado, P. Matos, V. Manquinho, and J. Marques-Silva, "Counting Models in Integer Domains," in *Proc. Int'l Conf. on Theory and Applications of Satisfiability Testing (SAT)*, A. Biere and C. P. Gomes, Eds. Springer, 2006, pp. 410–423.
[103] T. Toda and T. Soh, "Implementing Efficient All Solutions SAT Solvers," *ACM J. of Experimental Algorithmics (JEA)*, vol. 21, no. 1, pp. 1.12:1–1.12:44, 2016.
[104] E. Kuiter, C. Sundermann, T. Thüm, T. Heß, S. Krieter, and G. Saake, "How Configurable is the Linux Kernel? Analyzing Two Decades of Feature-Model History," *Trans. on Software Engineering and Methodology (TOSEM)*, 2025, to appear.
[105] M. H. t. Beek, K. Schmid, and H. Eichelberger, "Textual Variability Modeling Languages: An Overview and Considerations," in *Proc. Int'l Systems and Software Product Line Conf. (SPLC)*. ACM, 2019, pp. 151–157.
[106] J.-M. Horcas, M. Pinto, and L. Fuentes, "A modular metamodel and refactoring rules to achieve software product line interoperability," vol. 197, Mar. 2023.
[107] E. G. Karpenkov, K. Friedberger, and D. Beyer, "JavaSMT: A Unified Interface for SMT Solvers in Java," in *Proc. IFIP Working Conf. on Verified Software: Theories, Tools, Experiments (VSTTE)*. Springer, 2016, pp. 139–148.
[108] D. Chistikov, R. Dimitrova, and R. Majumdar, "Approximate counting in smt and value estimation for probabilistic programs," in *Proc. Int'l Conf. on Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*. Berlin, Heidelberg: Springer, 2015, pp. 320–334.
[109] L. De Moura and N. Bjørner, "Satisfiability Modulo Theories: An Appetizer," in *Proc. Brazilian Symposium on Formal Methods (SBFM)*. Springer, 2009, pp. 23–36.
[110] T. Thüm, S. Apel, C. Kästner, I. Schaefer, and G. Saake, "A Classification and Survey of Analysis Strategies for Software Product Lines," *ACM Computing Surveys (CSUR)*, vol. 47, no. 1, pp. 6:1–6:45, 2014.
[111] R. Heradio, D. Fernández-Amorós, J. A. Cerrada, and I. Abad, "A Literature Review on Feature Diagram Product Counting and Its Usage in Software Product Line Economic Models," *Int'l J. Software Engineering and Knowledge Engineering (IJSEKE)*, vol. 23, no. 08, pp. 1177–1204, 2013.
[112] M. Varshosaz, M. Al-Hajjaji, T. Thüm, T. Runge, M. R. Mousavi, and I. Schaefer, "A Classification of Product Sampling for Software Product Lines," in *Proc. Int'l Systems and Software Product Line Conf. (SPLC)*. ACM, 2018, pp. 1–13.