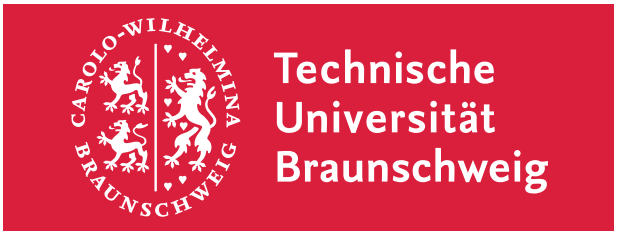

Master's Thesis

Applications of #SAT Solvers for Product Lines

Author:

Chico Sundermann

July 22, 2020

Advisors:

Prof. Dr.-Ing. Ina Schaefer
Michael Nieke, M. Sc.

TU Braunschweig
Institute of Software Engineering and Automotive Informatics

Prof. Dr.-Ing. Thomas Thüm

University of Ulm
Institute of Software Engineering and Programming Languages

https://doi.org/10.24355/dbbs.084-202009161329-0

Sundermann, Chico:
Applications of #SAT Solvers for Product Lines
Master's Thesis, TU Braunschweig, 2020.

https://doi.org/10.24355/dbbs.084-202009161329-0

Abstract

Product lines are widely used for managing families of similar products. Typically, product lines are complex and infeasible to analyze manually. In the last two decades, product-line analyses have been reduced to satisfiability problems which are well understood. However, there are methods for which satisfiability is not sufficient. Recently, researchers begun to reduce other problems to #SAT. Yet, only few applications have been considered and those are fairly limited in their scope. Furthermore, the authors mainly propose ad-hoc solutions that are only applicable under certain restrictions or do not scale to large product lines. In this thesis, we aim show the benefits of applying #SAT for the analysis of product lines. To this end, we make the following contributions: First, we summarize applications dependent on #SAT considered in the literature and propose new applications to motivate the usage of #SAT technology. Second, we present a variety of algorithms and optimizations for these applications including new proposals. Third, we empirically evaluate 10 proposed algorithms with 14 off-the-shelf #SAT solvers on 131 industrial feature models to identify the fastest algorithms and solvers. Our results show that for each analysis at least one algorithm and solver scale on a vast majority of the feature models, whereas Linux and an automotive model not be analyzed at all. In addition, our results further reveal the benefits of knowledge compilation to deterministic decomposable negation normal form for performing counting-based analyses. Overall, our work shows that #SAT dependent analyses for feature models open a new variety of different applications and scale to a large number of industrial feature models.

https://doi.org/10.24355/dbbs.084-202009161329-0

https://doi.org/10.24355/dbbs.084-202009161329-0

Zusammenfassung

Produktlinien sind weit verbreitet für die Verwaltung von Familien verwandter Produkte. In der Regel sind Produktlinien komplex und manuell schwer zu analysieren. In den letzten zwei Jahrzehnten wurden Produktlinienanalysen auf Erfüllbarkeitsprobleme reduziert, für welche es eine Vielzahl an effizienten Werkzeugen gibt. Allerdings ist Erfüllbarkeit nicht für alle Analysen hinreichend. Kürzlich haben Forscher damit begonnen, andere Probleme auf #SAT zu reduzieren. Es wurden jedoch nur wenige Anwendungen in Betracht gezogen und auch der Anwendungsbereich ist begrenzt. Darüber hinaus schlagen die Autoren hauptsächlich Ad-hoc-Lösungen vor, die nur unter bestimmten Einschränkungen der Produktlinien anwendbar sind oder nicht für große Produktlinien skalieren. In dieser Arbeit zeigen wir die Vorteile von #SAT Anwendungen für Produtlinien auf. Unser wissenschaftlicher Beitrag besteht aus den folgenden drei Punkten: Zuerst fassen wir die in der Literatur betrachteten #SAT-Anwendungen zusammen und schlagen neue Anwendungen vor, um den Einsatz von #SAT-Technologien zu motivieren. Zweitens stellen wir eine Vielzahl von Algorithmen und Optimierungen für diese Anwendungen vor, einschließlich neuer Vorschläge. Drittens führen wir eine empirische Evaluation von 10 der vorgeschlagenen Algorithmen mit 14 #SAT-Solvern auf 131 industriellen Feature-Modellen aus, um die schnellsten Algorithmen und Solver zu identifizieren. Die Ergebnisse der Evaluation zeigen, dass wir für jede Analyse wenigstens einen Algorithmus und Solver identifiziert haben, die für industrielle Feature-Modelle skalieren. Dazu sind die Ergebnisse ein starker Indikator für die Vorteile des Einsatzes von d-DNNFs bei #SAT-Anwendungen. Insgesamt zeigt unsere Arbeit, dass #SAT-abhängige Analysen für Feature-Modelle eine Vielzahl neuer unterschiedlicher Anwendungen ermöglicht und für viele industirelle Feature-Modelle skaliert.

https://doi.org/10.24355/dbbs.084-202009161329-0

https://doi.org/10.24355/dbbs.084-202009161329-0

Contents

https://doi.org/10.24355/dbbs.084-202009161329-0

https://doi.org/10.24355/dbbs.084-202009161329-0

1. Introduction

Mass customization allows developing individualized products while maintaining low development costs. A common way to achieve mass customization are product lines which represent a family of similar products [BSRC10]. The products are decomposed into subsets called features [SAKS16]. Composing them allows their reuse across multiple products [BSRC10]. Each different product corresponds to a unique combination of features, called configuration. A common way to specify the feature set and valid configurations of a product line is a feature model [CW07, Bat05]. This describes the underlying product line by limiting the set of valid configurations [BSRC10, Bat05, CW07].

Checking by hand, whether a given configuration satisfies all constraints specified by the feature model, is not trivial and time-consuming [BSRC10]. Industrial feature models often contain thousands of features and constraints [STS20, KZK10]. Keeping track of every single dependency between features manually is therefore expensive and prone to errors. It follows that automated support is required [Bat05]. Further examples of important analyses that suffer the same issue are checking whether a feature model induces any valid configurations, or a feature is dead (i.e., is not part of any valid configuration) [BSRC10, SSK+20]. To automate such analyses tools typically rely on satisfiability-based solvers like SAT solvers [Bat05, SKT+16, MWC09, PLP11, CW07, PSK+10, Seg08, GAT+16], other constraint satisfaction problem (CSP) solvers [BTRC05, PLP11, Seg08, WBS+10], and binary decision diagrams (BDD) [MWCC08, PLP11, Seg08, GAT+16, CW07].

For some analyses of feature models, satisfiability is not sufficient. For example, uniform random sampling is reliant on counting all valid configurations [OGB+19, OBMS17, MOP+19]. In theory, it is possible to count solutions with regular SAT, CSP, and BDDs [PLP11]. However, this typically comes with major scalability issues [PLP11, TS16, OGB+19]. #SAT solvers are specifically optimized for counting solutions and made significant advances in the last decade [BJP00, Thu06, BSB15, Dar04, LM17].

Research is still required regarding the scalability and possibilites of #SAT solvers for the analysis of product lines. Currently, only few applications are considered in

https://doi.org/10.24355/dbbs.084-202009161329-0

the literature. #SAT had major scalability issues when applied to feature models in previous surveys [PLP11, KZK10]. This is probably a main reason for the limited research. However, early regular SAT solvers were significantly slower than modern ones [MFM04]. Now, state-of-the-art regular SAT solvers can even analyze large scale feature models in a short amount of time [LGCR15]. Thus, it is reasonable to assume that #SAT solvers will keep evolving and continually scale to larger feature models.

For applications dependent on counting the number of valid configurations, the literature often only presents the application without providing a feasible algorithm to compute it [HFACA13, CMC05, BSRC10]. Other works only provide solutions for simplified feature models (e.g., feature models without cross-tree constraints) [FAGS09, HGFACC11]. Our main contributions are presenting applications dependent on #SAT, providing implementations, and examining their scalability.

Goal of this Thesis

In this thesis, we aim to identify applications for #SAT and examine the scalability of these using off-the-shelves #SAT solvers. Explicitly, we aim to answer the following research questions during the thesis:

- *RQ1: Which applications for #SAT on product lines are considered in the literature?*
- *RQ2: What are new applications for #SAT on software product lines?*
- *RQ3: For a given #SAT application, is there an algorithm that scales to industrial product lines?*
- *RQ4: For a given algorithm, what is the fastest off-the-shelf #SAT solver?*
- *RQ5: What is the performance of approximate #SAT solvers for analyzing product lines?*

We answer the first three research questions *RQ1/2* theoretically and the following three research questions *RQ3-5* with an empirical evaluation.

To answer *RQ1*, we summarize applications for product lines that are dependent on counting valid configurations already considered in the literature. To answer *RQ2*, we propose new applications. The resulting survey indicates the relevance of #SAT in the product line domain and presents potential benefits of using the technique.

To answer *RQ3*, we examine the scalability of algorithms that can be used to compute results for the applications in the survey. Algorithms are only usable in practice if they scale to real product lines. Thus, we present empirical evaluations of the algorithms on industrial feature models. The results are used to identify scalable algorithms and compare different optimizations.

To answer *RQ4*, we compare different off-the-shelves #SAT solvers with an empirical evaluation. The scalability of the algorithms heavily depends on the used #SAT solver. There are many #SAT solvers considered in the literature [Dar04,

https://doi.org/10.24355/dbbs.084-202009161329-0

LM17, MMBH10, BJP00, SBK05a, Thu06, BSB15, Bie08, KLMT13, KMM13, TS16, OD15]. Our previous results showed that their performance differs significantly for counting the number of valid configurations [STS20]. A comparison of different solvers yields the following two potential advantages. On the one hand, the solvers that perform well in general can be identified. On the other hand, some solvers may perform better for some algorithms but worse for others. We aim to identify the most promising solver for a specific algorithm.

To answer *RQ5*, we evaluate the runtime of using approximate #SAT solvers. Approximate #SAT solvers estimate the number of satisfying assignments of a propositional formula within a given confidence level [AT17, GSS06, BG19]. Using approximate #SAT solvers may enable applications on feature models for which exact results are infeasible.

Structure of the Thesis

In Chapter 2, we provide the background on which the rest of this thesis is based on and we describe the state-of-the-art of feature model analysis. In Chapter 3, we give a survey of possible #SAT applications for product lines to answer *RQ1* and *RQ2*. This survey consists of applications considered in the literature and our own proposals. In Chapter 4, we discuss underlying algorithms and optimizations for the applications. In Chapter 5, we provide implementations for the algorithms. Furthermore, we describe the integration of #SAT solvers and their applications to FeatureIDE [fea19], a popular environment for feature-based development. In addition, we illustrate the the implementation of our evaluation framework. In Chapter 6, we evaluate the applications using the implemented algorithms and several experiments. The results are used to answer RQ3-5. In Chapter 7, we present work that is related to ours. In Chapter 8, we summarize the insights of the thesis and provide a short conclusion. In Chapter 9, we discuss possible future work.

https://doi.org/10.24355/dbbs.084-202009161329-0

https://doi.org/10.24355/dbbs.084-202009161329-0

2. Background

In this chapter, we provide the required basics to understand the following chapters of this thesis. First, we describe the basics of SAT and #SAT and the differences between the two problems. Second, we present the necessary background for feature models and how SAT is used to analyze them.

2.1 SAT

The SAT problem corresponds to the satisfiability of propositional formulas. Each propositional formula F holds a set of variables $vars(F)$ and consists of literals $l \in vars(F)$ and logical operators $\neg$ (not), $\wedge$ (conjunction), $\vee$ (disjunction), $\Rightarrow$ (implies), $\Longleftrightarrow$ (equals) that connect the literals [BSRC10]. For every variable, a truth value $\{\top, \bot\}$ can be assigned. $\top$ corresponds to true and $\bot$ to false. Let A be the set of possible assignments. An assignment $\alpha \in A$ is a function $\alpha : vars(F) \rightarrow \{\bot, \top, UNDEF\}$ that maps variables to truth values [KZK10]. If every variable is mapped to either $\top$ or $\bot$, we call the assignment full. Otherwise, we call it a partial assignment [KZK10]. An assignment α that satisfies a propositional formula F is denoted by $\alpha(F) = \top$. If α does not satisfy F, $\alpha(F) = \bot$ holds. $|\alpha|$ corresponds to the number of variables that are assigned either $\top$ or $\bot$ in α. A formula F that is a tautology (i.e., $\forall \alpha \in A : \alpha(F) = \top$) or a contradiction (i.e., $\forall \alpha \in A : \alpha(F) = \bot$) is denoted by $F \equiv \top$ or $F \equiv \bot$, respectively. SAT is used in many areas and is the most popular NP-complete problem [NOT06].

2.1.1 Forms of Propositional Formulas

Different formats of propositional formulas are considered in the literature [DM02, BHvM09, MMBH10]. Those typically have different advantages (e.g., fast queries, succinctness, readability [DM02]) and can be translated into each other without changing the semantic of the formula. For a better understanding of the different formats, we display formulas as directed acyclic graphs (DAG). Each node of the diagram is a logical operator or a literal. For example, $(A \wedge B) \vee \neg C$ can be represented by the DAG displayed in Figure 2.1.

https://doi.org/10.24355/dbbs.084-202009161329-0

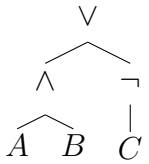

Figure 2.1: DAG representing $(A \wedge B) \vee \neg C$

Negation Normal Form

In a formula F in negation normal form (NNF) negations may only appear directly in front of literals, in contrast to in front of subformulas consisting of logical operators and literals [HR04]. Every propositional formula can be translated to NNF using de Morgan Rules [HR04].

Definition 2.1 (Negation Normal Form)**.** *A propositional formula F is in* NNF *iff every negation symbol appears directly in front of a literal [DM02].*

Figure 2.2 shows a formula that is not in NNF as there is a negation symbol in front of $B \vee C$. However, the formula can easily be translated to NNF.

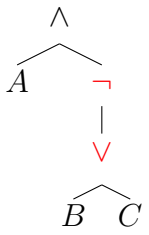

Figure 2.2: Formula not in NNF

Figure 2.3 shows a formula that is semantically equivalent to Figure 2.2, but it is translated to NNF using de Morgan rules. The translation to NNF is possible for every propositional formula. NNF is a superset of many prominent propositional formula formats (e.g. conjunctive normal form) [DM02].

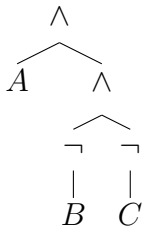

Figure 2.3: Formula in NNF

Conjunctive Normal Form

Conjunctive normal form (CNF) is a subset of NNF [DM02]. Modern SAT solvers are typically based on CNFs [TBW04]. One of the most promiment algorithms to solve SAT problems is Davis-Putnam-Logemann-Loveland (DPLL) which we describe in Section 2.1.2. DPLL based solvers typically use CNFs as input [Lib00, HR04]. In

https://doi.org/10.24355/dbbs.084-202009161329-0

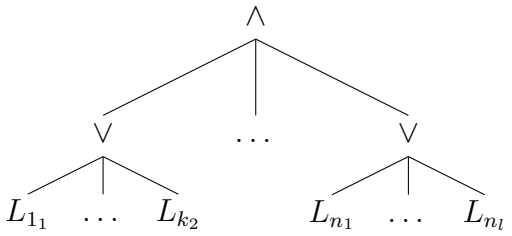

Figure 2.4: Formula in CNF

addition to the NNF property, a CNF holds the following characteristics. A CNF is a conjunction $C = D_1 \wedge ... \wedge D_n$ of arbitrarily many disjunctions $D_i = L_{i_1} \vee ... \vee L_{i_m}$ of different literals L_{i_j}. Formally, a CNF holds the following two properties. First, a CNF is flat (i.e., the maximum nesting level is two). Second, every disjunction is a clause (i.e., the child nodes of a disjunction are distinct and either positive or negative literals) [DM02]. Figure 2.4 displays the DAG of a CNF formula.

Definition 2.2 (Flatness)**.** *A propositional formula F is considered flat iff the maximum nesting level in F is at most 2 [DM02].*

Definition 2.3 (Conjunctive Normal Form)**.** *A propositional formula F is in CNF iff F is an NNF, flat, and every disjunction is a clause [DM02].*

Any propositional formula can be translated to CNF using equivalence rules. However, this potentially increases the size of the formula exponentially [OGB+19]. Instead, equisatisfiable transformations are typically used [MOP+19, OGB+19, DDM06]. Two formulas F and F' are considered equisatisfiable if both are satisfied and unsatisfied for the same assignments (i.e., $(\alpha(F) = \top \iff \alpha(F') = \top) \wedge (\alpha(F) = \bot \iff \alpha(F') = \bot)$). One example, for a popular equisatisfiable transformation is Tseitins transformation [Tse83].

Decomposable Deterministic Negation Normal Form

The decomposable deterministic negation normal form (d-DNNF) is a subset of NNF that holds the properties decomposable and deterministic [DM02]. In contrast to CNF, a d-DNNF is not flat [DM02]. Formulas in d-DNNF allow a high number of queries in polynomial time (e.g., model counting) [DM02]. Furthermore, any CNF can be translated to d-DNNF in theory [LM17]. In the following, we define the properties of a d-DNNF and show examples of formulas that fulfill them.

Definition 2.4 (Deterministic)**.** *A propositional formula F is deterministic iff the children $D_1, ..., D_n$ of each disjunction in F share no common solutions (i.e., $\forall i, j, i \neq j : D_i \wedge D_j \equiv \bot$) [DM02].*

Figure 2.5 shows a DAG that represents a formula that is not deterministic as the children of the root $\vee$ share the solution {A,B}. Figure 2.6 shows an example of a DAG for a deterministic formula, as $A \wedge (\neg A \wedge B) \equiv \bot$.

Definition 2.5 (Decomposable)**.** *A propositional formula F is decomposable iff the children $C_1, ..., C_n$ of each conjunction in F share no variables(i.e., $\forall i, j, i \neq j : C_i \cap C_j = \emptyset$) [DM02].*

https://doi.org/10.24355/dbbs.084-202009161329-0

Figure 2.5: Not deterministic formula

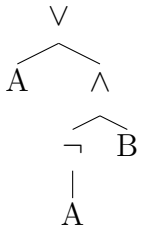

Figure 2.6: Deterministic formula

Figure 2.7 shows a DAG that represents a non-decomposable formula, as the children of the root $\wedge$ share the variable A. Figure 2.8 shows a DAG representing a decomposable formula, as A and $B \wedge \neg C$ share no variables.

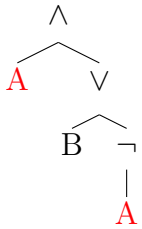

Figure 2.7: Not decomposable formula

The children of a disjunction may contain different sets of variables in a d-DNNF. This makes some analyses (e.g., counting the number of satisfying assignments) more complex [Dar01a]. A formula that is smooth, contains no disjunctions whose child nodes hold distinct variables. Adapting a d-DNNF so that it is smooth has a polynomial time complexity [Dar01a]. Figure 2.9 shows a formula that is not smooth as the children of the disjunction contain different variables. Figure 2.10 shows an equivalent formula that is smooth. Let F_1, F_2 be children of an Or node and v a variable that is part of F_1 but not of F_2. The Or node can be smoothed by replacing F_2 with $F_2 \wedge (v \vee \neg v)$. This does not change the semantics of the formula preserves the properties decomposable and deterministic. Applying this technique for every Or-node results in a smooth d-DNNF.

Definition 2.6 (Smooth)**.** *A propositional formula F is smooth, iff the children $D_1, ..., D_n$ of each disjunction in F contain the same variables(i.e., $\forall i, j : vars(D_i) = vars(D_j)$) [DM02].*

A special case of d-DNNFs considered in the literature are Decision-DNNFs [LM17, Dar02]. In Decision-DNNFs, Or-nodes are replaced with decision nodes. Each decision node has a left n_l and a right n_r child and corresponds to one variable v. It can be interpreted as **if** v **then** n_l **else** n_r. In propositional logic, a decision node is equivalent to $(v \wedge n_l) \vee (\neg v \wedge n_r)$. It is easy to see that this expression is deterministic

https://doi.org/10.24355/dbbs.084-202009161329-0

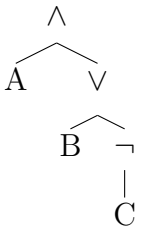

Figure 2.8: Decomposable formula

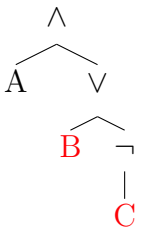

Figure 2.9: Not smooth formula

in any case and also decomposable if n_l and n_r share no variables. Thus, decision nodes do not violate the restrictions of a d-DNNF and a Decision-DNNF is also a d-DNNF.

Knowledge compilation is about translating formulas into a format that supports more or other types of queries in polynomial time [DM02]. d-DNNF is a promising target language as there are algorithms exploiting the properties of d-DNNFs that, inter alia, count the number of satisfying assignments in polynomial time [DM02]. Current transformation algorithms typically translate a CNF to d-DNNF with a traversal similar to DPLL [Dar02, MMBH10]. Another well-known format that supports even more polynomial time queries are binary decision diagrams. However, d-DNNFs are known to be more succinct [Dar02, DM02]. Thus, they should scale to larger formulas than BDDs.

Binary Decision Diagram

Binary decision diagrams (BDDs) are another target language of knowledge compilation [DM02]. Intuitively, a BDD encodes every possible assignment of a propositional formula as paths of a DAG. Each node of a path corresponds to a variable. The path of a satisfying assignment α ends in a $\top$-node to indicate $\alpha(F) = \top$.

Formally, a BDD consists of two types of nodes, namely variable and terminal nodes. Each BDD has two of the latter corresponding to the truth values $\top$ and $\bot$, respectively [HR04, HSJ+04]. Each variable node corresponds to one variable $v \in vars(F)$. Furthermore, each variable node has one low and one high edge leading to another node. Let F be a propositional formula represented by the BDD and $\alpha(F)$ be an assignment. At a variable node N_v corresponding to $v \in vars(F)$, the low/high edge is taken if $\alpha(v) = \bot/\top$. Traversing the BDD with this logic ends in the $\top$ or $\bot$ terminal node. If the traverse corresponding to α ends in $\top$, it follows that $\alpha(F) = \top$. Otherwise, it is $\alpha(F) = \bot$ [HR04, HSJ+04].

https://doi.org/10.24355/dbbs.084-202009161329-0

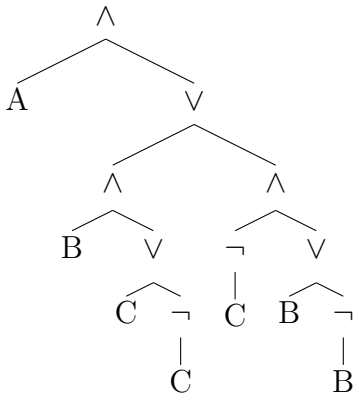

Figure 2.10: Smooth formula

2.1.2 Ways to Solve SAT Problems

Davis–Putnam–Logemann–Loveland (DPLL) is the most prominent algorithm used to check whether a propositional formula is satisfiable [Lib00]. It is a depth-first-search backtracking algorithm that assigns a variable at each step. The algorithm typically operates on a CNF (i.e., on a conjunction of clauses). After each assignment, the algorithm checks whether all clauses are satisfied or at least one cannot be satisfied anymore. If all clauses are satisfied, the current set of assignments is returned as solution. If a clause cannot be satisfied given the current assignment, the algorithm backtracks [Lib00]. The actual algorithm is shown in Algorithm 1.

Algorithm 1 DPLL(F,α)

if $\alpha(F) = \top$ **then**
  return $\top$
end if
if $\alpha(F) = \bot$ **then**
  return $\bot$
end if
l_{next} := `getNextUnassignedVariable()`
$\alpha_{next} := \alpha \cup \{l_{next}\}$
if $dpll(F, \alpha_{next}) = \top$ **then**
  return $\top$
end if
$\alpha_{\neg next} := \alpha \cup \{\neg l_{next}\}$
if $dpll(F, \alpha_{\neg next}) = \top$ **then**
  return $\top$
end if
return $\bot$

There are many other possible algorithms to compute the satisfiability of a propositional formula (e.g. natural deduction [HR04], local search algorithms [HS00], and the marking algorithm for horn formulas [HR04]). Another one that is relevant for this thesis is exploiting the properties of knowledge compilation target languages (e.g., d-DNNF and BDD) [DM02]. Here, the main effort lies in the translation to the

https://doi.org/10.24355/dbbs.084-202009161329-0

target language [Dar02, DM02]. For example, a propositional formula is satisfiable iff the BDD representing it contains a path to the $\top$ terminal node [HR04].

2.1.3 Complexity

SAT is NP-complete [Joh92]. This implies two properties. First, a solution for SAT can be verified with a polynomial time complexity [Wel82]. Second, assuming $P \neq NP$, there is no deterministic algorithm that finds a solution with a polynomial time complexity [Wel82]. However, various heuristics have been considered in the literature that scale to large formulas in practice [GPFW96].

2.2 #SAT

In contrast to regular SAT, #SAT corresponds to the number of satisfying assignments of a propositional formula [GSS06, KZK10, BDP03]. Let F be a formula and A_F the set of all possible assignments for the variables $vars(F)$. Formally, a #SAT solver computes the cardinality of the set of satisfying full assignments $\#F = |\{\alpha \in A_F | \alpha(F) = \top\}|$.

2.2.1 Ways to Solve #SAT Problems

A naive solution to compute the number of satisfying assignments is enumerating them using regular SAT solvers. First, the solver computes a full satisfying assignment α, which can be translated to a term. The term is created by conjuncting a literal l_i for all variables $f_i \in vars(F)$: $l_1 \wedge ... \wedge l_n$. If $\alpha(f_i) = \bot$, the corresponding literal l_i is negated. The resulting term is negated and conjuncted to F. Negating a term with the De Morgan rule (e.g., $\neg(A \wedge B) \equiv \neg A \vee \neg B$) results in a clause c [HR04]. Thus, the resulting formula $F' = F \wedge c$ is still in CNF. In the next step, F' is given as input to a SAT solver. As α is not a satisfying assignment for F', another assignment is returned. This procedure can be repeated until there is no satisfying assignment left to compute the number of satisfying assignments [TS16].

Another possible way is to adapt the DPLL procedure described in Algorithm 1. Let F be a propositional formula in CNF with $n = |vars(F)|$. If a satisfying assignment α is found, regular DPLL stops the traversal and returns α. The adaption for #SAT computes the number of satisfying assignments of the current branch using the $n - |\alpha|$ freely assignable variables. Each of those free variables can either be assigned $\top$ or $\bot$. Thus, the number of satisfying assignments is $2^{n-|\alpha|}$. Then, the procedure backtracks to find remaining satisfying branches. By definition of the DPLL procedure, the assignments of the different branches are distinct. Thus, the number of all satisfying assignments is the sum of the number of assignments of the branches. [HFACA13, BHvM09, BDP03]. Adaptations of DPLL for #SAT is used by many state-of-the-art #SAT solvers [BJP00, SBK05a, Thu06, BSB15, Bie08, KMM13]. Algorithm 2 shows a basic DPLL algorithm that computes the number of satisfying assignments.

A d-DNNF can be used to compute the number of satisfying assignments in polynomial time [DM02]. The idea is to exploit the properties deterministic and decomposable of a d-DNNF. The children $D_1, ..., D_n$ of a disjunction D in a deterministic

https://doi.org/10.24355/dbbs.084-202009161329-0

Algorithm 2 `CountDPLL`(F, α)

```
1: if α(F) = 1 then
2:    return 2^{n−|α|}
3: end if
4: if α(F) = 0 then
5:    return 0
6: end if
7: l_next := getNextUnassignedVariable()
8: count_next := CountDPLL(F, α ∪ l_next)
9: count_¬next := CountDPLL(F, α ∪ l_¬next)
10: return count_next + count_¬next
```

formula share no common solutions [DM02]. Thus, the number of satisfying assignments of the children can just be summed up (i.e.,$\#D = \sum_{i=1}^{n} \#D_i$) [BHvM09]. However, this requires some extra handling of variables that are not part of every child if the d-DNNF is not smooth. As an example, consider the deterministic disjunction $F = (A \wedge B) \vee (\neg A \wedge C)$. Without the context of the entire propositional formula, $A \wedge B$ and $\neg A \wedge C$ both induce one solution each. However, F induces four solutions which is more than $1 + 1 = 2$ solutions implied by the formula for disjunctions. The problem is that B and C do not appear in every child of the disjunction. We consider two possible ways to acquire correct results. First, we can adapt the formula to consider missing variables. For each child node, every missing variable can be either $\top$ or $\bot$. Thus, the model count needs to be multiplied with two for every variable. Let M_i be the set of variables that appear in a child $D_{j \neq i}$ but not in D_i. To get correct results the formula for disjunctions can be adapted as follows: i.e.,$\#D = \sum_{i=1}^{n} \#D_i * 2^{|M_i|}$. Second, we can smooth the d-DNNF. Then, each child of a disjunction contains the same set of variables and we do not need to handle the issue for each single query on the d-DNNF. Smoothing our example results in $F' = (A \wedge B \wedge (C \vee \neg C)) \vee (\neg A \wedge C \wedge (B \vee \neg B))$. Both children of the disjunction in F' induce two solutions without even without the context of F'. Thus, the adaption to handle missing variables is not required as $2 + 2 = 4$ is correct. The children $C_1, ..., C_m$ of a conjunction C in a decomposable formula share no variables [DM02]. Thus, the number of satisfying assignments of the children can just be multiplied (i.e.,$\#C = \prod_{i=1}^{m} \#C_i$) [BHvM09]. Each leaf is a literal and contains one satisfying assignment. Applying the rules for deterministic and decomposable nodes, the number of satisfying assignments $\#F$ can be computed with a single traversal of d-DNNF [BHvM09]. Algorithm 3 shows the actual algorithm for a smooth d-DNNF.

Computing the number of satisfying assignments with a BDD is possible in polynomial time, like with d-DNNF [BHvM09]. Starting from the $\top$ terminal node, every path can be traversed to compute all satisfying assignments. The paths starting from the $\bot$ terminal node do not imply any satisfying assignment. Thus, a traversal of these can be omitted to save time [BHvM09].

#SAT solvers typically operate on a CNF [Dar04, LM17, MMBH10, BJP00, SBK05a, Thu06, BSB15, Bie08]. For a correct result, an equisatisfiable transformation is not sufficient as it is also necessary to preserve the number of satisfying assign-

https://doi.org/10.24355/dbbs.084-202009161329-0

Algorithm 3 `recurseDDNNF`(*node*)

```
1: if node.isLeaf() then
2:    return 1
3: end if
4: if node.isAnd() then
5:    result := 1
6:    for child in node.getChildren() do
7:       result := result * recurseDDNNF(child)
8:    end for
9:    return result
10: end if
11: if node.isOr() then
12:    result := 0
13:    for child in node.getChildren() do
14:       result := result + recurseDDNNF(child)
15:    end for
16:    return result
17: end if
```

Algorithm 4 `countDDNNF`(F)

```
1: return recurseDDNNF(ROOT_F)
```

ments [OGB+19]. Some transformations introduce new variables to simplify the transformation [PG86, Tse83]. In this case, it is possible that the resulting formula is equisatisfiable but contains more solutions resulting from the added variables [OGB+19]. This is an important point to consider for #SAT to prevent wrong results.

2.2.2 Complexity

#SAT is widely assumed to be a harder problem than regular SAT [BSB15, BG19]. It is obvious that #SAT is at least as hard as SAT. Checking whether there is at least one solution is trivial after computing the number of solutions [Joh92]. Furthermore, there are problems which can be solved in polynomial time for SAT but not for #SAT (e.g., horn-clauses or 2-CNF) [BG19]. While SAT is an NP-problem, #SAT is #P-complete [BHvM09]. Intuitively, #P problems correspond to computing the number of solutions for an instance of a problem that are decidable in polynomial time complexity [BHvM09, KZK10]. Every #P-complete problem can be reduced to every other #P-complete problem in polynomial time [BHvM09].

2.3 Feature Models

A product line describes a family of similar products in contrast to standalone products [BSRC10]. Typically, these products can be decomposed into features, which are reused in multiple products [SAKS16]. Product lines enable cost-efficient development while still allowing individualized products [PLP11].

https://doi.org/10.24355/dbbs.084-202009161329-0

Feature models are commonly used to describe a product line in terms of features and dependencies between them on an abstract level [PLP11, BSRC10]. The dependencies consist of hierarchical relations between a parent feature and its child, and cross-tree constraints [BSRC10].

Definition 2.7 (Feature Model)**.** *A feature model* $FM = (FEATS, ROOT, \omega, REL, P, CTC)$ *is a 6-tuple with:*

- $FEATS$ *is the set of features.*
- $ROOT$ *is the root of the feature model.*
- $P : FEATS \rightarrow FEATS$ *is a function that maps each feature to its parent. We define* $CHILDREN_p = \{f \in FEATS | P(f) = p\}$ *as the set of children of feature* p*. A child can only be selected if its parent is selected.*
- $\omega : FEATS \rightarrow \{mandatory, optional\}$ *is a function that maps each feature to either mandatory or optional. A mandatory feature always needs to be selected if the features parent is selected. This is not the case for an optional feature.*
- $REL : FEATS \rightarrow \mathbb{N} \times \mathbb{N}$ *is a function that describes the relation of a feature to its children.* $REL(f) =< n, m >$ *indicates that at least* n *and at most* m *children of* f *have to be selected if* f *is selected.*
- CTC *is the set of cross-tree constraints. Each constraint* $ct \in CTC$ *is in propositional logic with features as variables.*

A feature diagram is often used to visualize a feature model [HFACA13]. Figure 2.11 displays the feature diagram of a simplified car. Each car requires a *Carbody* denoted by a Mandatory-flag and exactly one type of a *Gearbox* denoted by an Alternative-group. Additionally, the car may contain a *Radio* that can be further configured (e.g., it may contain *Bluetooth*). If *Ports* is selected, a car may have *USB*, *CD*, or both which is denoted by an Or-group. Overall, the following types of hierarchical relations are displayed.

- An *Alternative*-relation indicates that exactly one child of a feature p needs to be selected if p is selected (i.e., $REL(p) =< 1, 1 >$) [BSRC10]. The simplified car requires exactly one of *Manual* and *Automatic*. Thus, the features are alternative to each other.
- An *Or*-relation indicates that at least one of the n children of a feature p needs to be selected if p is selected (i.e., $REL(p) =< 1, n >$) [BSRC10]. If *Ports* is selected, the car requires one or both of *USB* and *CD*. Thus, the features are in an or-relation.
- An *And*-relation indicates that each child is either *Mandatory* or *Optional*. The children of an *Or* or an *Alternative* are not considered *Mandatory* or *Optional*.

https://doi.org/10.24355/dbbs.084-202009161329-0

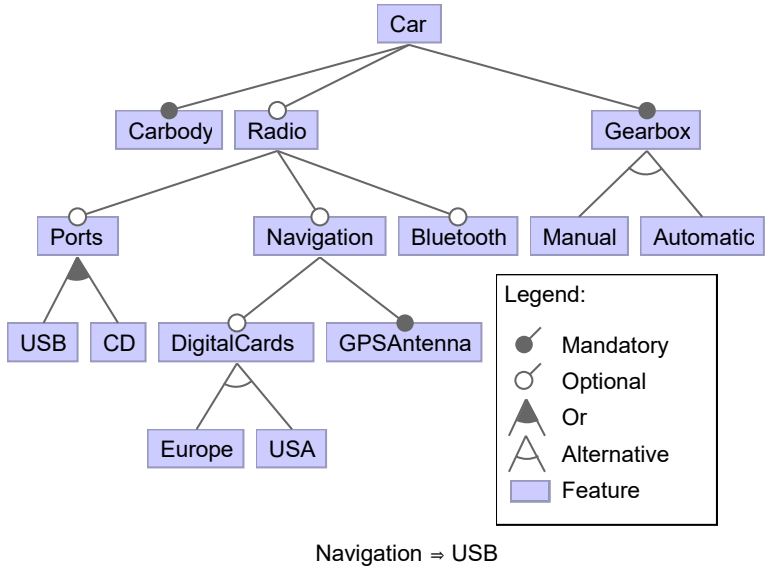

Figure 2.11: Simplified car adapted from Ananieva et al. [Ana16]

- An *Optional*-flag indicates that the feature f does not need to be selected if its parent is selected (i.e., $\omega(f) = optional$) [BSRC10]. A car may have a *Radio*, but does not require one. Thus, the feature is optional.
- A *Mandatory*-flag indicates that the feature f needs to be selected if its parent is selected (i.e., $\omega(f) = mandatory$) [BSRC10]. A car requires a *Carbody* in any case. Thus, the feature is mandatory.

For the remainder of this thesis, we limit the possible relations between a feature and its children to *Alternative* (i.e., $REL(p) =< 1, 1 >$), *Or* (i.e., $REL(p) =< 1, n >$), and *And*.

The feature diagram also displays one cross-tree constraint $Navigation \Rightarrow USB$. This means that every car that includes *Navigation* also requires *USB*. The hierarchical relations and the cross-tree constraints specify the set of valid configurations. Each feature model induces configurations that can be built by selecting and deselecting features.

Definition 2.8 (Configuration)**.** *A configuration* $C = (FM, I, E)$ *is a 3-tuple with:*

- $FM = (FEATS, ROOT, \omega, REL, P, CTC)$
- $I \subseteq FEATS$ *is the set of features included in the configuration* C*.*
- $E \subseteq FEATS$ *is the set of features excluded from the configuration* C*.*
- $I \cap E = \emptyset$*. A feature cannot be included and excluded in the same configuration* C *[BSRC10].*

Sometimes, one may aim to create a family of configurations, where some features are neither included or excluded. Such a configuration is called partial. All features of a full configuration are either included or excluded.

https://doi.org/10.24355/dbbs.084-202009161329-0

Definition 2.9 (Partial Configuration)**.** *A configuration* $C = (FM, I, E)$ *is partial iff* $I \cup E \subsetneq FEATS_{FM}$ *[BSRC10].*

Definition 2.10 (Full Configuration)**.** *A configuration* $C = (FM, I, E)$ *is full iff* $I \cup E = FEATS_{FM}$ *[BSRC10].*

There are $2^{|FEATS|}$ possible configurations [KZK10]. However, not all of them satisfy all constraints imposed by the feature model. We call a configuration that does not violate any properties of the feature model *valid* [BSRC10]. We define VC_{FM} as the set of valid configurations and $\#FM = |VC_{FM}|$ as the number of valid configurations of the feature model FM.

Definition 2.11 (Valid Configuration)**.** *A full configuration* $C = (FM, I, E)$ *is valid if each of the following properties is fulfilled.*

- $ROOT \in I$
- $\forall c \in FEATS : c \in I \Rightarrow P(c) \in I$
- $\forall c \in FEATS : P(c) = p, \omega(c) = mandatory : p \in I \Rightarrow c \in I$
- $\forall p \in FEATS : REL(p) =< n, m >: p \in I \Rightarrow n \leq |\{c \in (CHILDREN_p \cap I)\}| \leq m$
- $$\bigwedge_{ct \in CTC} ct \wedge \bigwedge_{i \in I} i \wedge \bigwedge_{e \in E} \neg e \equiv \top$$

It is not trivial to manually determine whether a given configuration is valid as every specified constraint has to be reviewed. Especially for larger feature models, it is time-consuming to check the validity of a configuration. Thus, automated support is required.

2.4 Feature Model Analysis

Manual analysis is error-prone and time-consuming, especially for larger feature models [BSRC10]. Automated analyses can be used to support developers and users of a feature model [BSRC10]. It can be used, inter alia, to extract information, detect anomalies, or accelerate product derivation [BSRC10].

It is difficult to keep track of all dependencies between the features of a feature model. Thus, edits may result in faulty models. Automated support can help developers to detect and resolve such problems. With cross-tree constraints, it is possible to create conflicts such that two or more constraints can never be fulfilled under the same assignment. In this case, the feature model does not induce any valid configuration. Such feature models are called *void* [BSRC10].

Definition 2.12 (Void model)**.** *A feature model* FM *is void iff* FM *induces no valid configurations.*

https://doi.org/10.24355/dbbs.084-202009161329-0

Not all design errors directly result in a void feature model. However, some may result in features that cannot be selected in any valid configuration. Such features are called *dead* [BSRC10].

Definition 2.13 (Dead Feature)**.** *A feature $f \in FEATS_{FM}$ is a dead feature iff there is no valid configuration (FM, I, E) with $f \in I$.*

A *core* feature is part of every valid configuration [BSRC10]. It may be beneficial to prioritize core features in the development, as no valid configuration can be built without them.

Definition 2.14 (Core Feature)**.** *A feature $f \in FEATS_{FM}$ is a core feature iff there is no valid configuration (FM, I, E) with $f \notin I$.*

Sometimes, the feature tree indicates that an illegal combination of features is valid. For example, a feature c may be modeled as optional but is there is no valid configuration that contains the parent of c but not c. In this case, c is a *false-optional* feature [BSRC10]. It behaves as a mandatory feature but is wrongfully modeled as optional.

Definition 2.15 (False-optional Feature)**.** *A feature c is false-optional iff $\omega(c) = optional$ and $\forall C = (FM, I, E) \in VC_{FM} : p \in I \Rightarrow c \in I$.*

A feature and its mandatory or false-optional feature always appear together in a valid configuration [Seg08]. Thus, they are part of the same atomic set. An atomic set is a set of features that only appears together in valid configurations. There is no valid configuration, that contains only a subset of features in an atomic set. Atomic sets can be regarded as a unit for the analysis of the feature model which may simplify the computation [Seg08, ZZM04].

Definition 2.16 (Atomic Set)**.** *Two features f, g are part of an atomic set A iff For every valid configuration $C = (FM, I, E) \in VC_{FM}$: $f \in I \iff g \in I$.*

Another use-case for feature model analysis is deriving valid configurations from a feature model. Manually, the user would have to keep track of all the dependencies to not accidentally create an invalid configuration. A given configuration should be checked for validity automatically. Another aspect is interactive support. For example, if a parent of a mandatory feature f is selected, f can automatically be selected as well [SSK+20].

SAT-based Analysis

SAT is commonly used to analyze feature models [HFACA13, BSRC10, Bat05]. Any feature model can be translated to a propositional formula using rules shown in Table 2.1 [MWC09, BSRC10]. The cross-tree constraints are already in propositional logic and can be conjuncted to the resulting formula.

We define F_{FM} as the propositional formula representing feature model FM. A configuration $C = (FM, I, E)$ can also be translated to propositional logic. We define $F_C = F_{FM} \wedge \bigwedge_{i \in I} \wedge \bigwedge_{e \in E} \neg e$ as the propositional formula representing C. A SAT solver can use such formulas F_{FM} and F_C to analyze feature models and configurations. Let $A \subset FEATS_{FM}$ be an atomic set.

https://doi.org/10.24355/dbbs.084-202009161329-0

Relation	Propositional Formula
Or	$p \iff o_1 \vee ... \vee o_n$
Alternative	$a_1 \iff (\neg a_2 \wedge ... \wedge \neg a_n \wedge p) \wedge$
	$a_2 \iff (\neg a_1 \wedge \neg a_3 \wedge ... \wedge \neg a_n \wedge p) \wedge$
	$... \wedge a_n \iff (\neg a_1 \wedge ... \wedge \neg a_{n-1} \wedge p)$
Optional	$c \Rightarrow p$
Mandatory	$c \iff p$
$ct \in CTC$	ct

Table 2.1: Translation Feature Model to Propositional Logic

Analysis	Propositional Computation
FM is void	F_{FM} is unsatisfiable
$f \in FEATS_{FM}$ is dead	$F_{FM} \wedge f$ is unsatisfiable
$f \in FEATS_{FM}$ is core	$F_{FM} \wedge \neg f$ is unsatisfiable
$c \in FEATS_{FM}$ is false-optional, $P(c) = p$	$F_{FM} \wedge p \wedge \neg c$ is unsatisfiable
$f, g \in A$ (Atomic Set)	$F_{FM} \wedge f \wedge \neg g$ and
	$F_{FM} \wedge g \wedge \neg f$ are unsatisfiable
C is valid configuration	F_C is satisfiable
Propagate selection of $f \notin I_C$ for C	$F_C \wedge \neg f$ is unsatisfiable

In the literature, most analyses are only based on regular SAT. However, F_{FM} can also be used as an input for a #SAT solver. Given F_{FM} as input, the solver computes the number of valid configurations of FM. Analyses that use #SAT solvers are presented in Chapter 3.

https://doi.org/10.24355/dbbs.084-202009161329-0

3. Applications

In this chapter, we discuss applications that depend on the number of valid configurations of a product line. Hereby, we present applications that are already considered in the literature and also provide new ones. This is supposed to answer *RQ1: Which applications are possible for #SAT in the product line domain?* and motivate the usage of #SAT solvers for product line analysis.

While relevant analyses exist that require counting the number of valid configurations (e.g., uniform random sampling [OBMS17, OBMS16, OGB+19, MOP+19]), only little research addressing this topic has been conducted. A major issue has been the scalability of counting the number of valid configurations [PLP11]. However, our prior results showed that it is feasible to use current #SAT solvers on many industrial systems [STS20]. Thus, it is possible to use them to enable new applications for product lines that are potentially relevant for industrial usage. The presented applications rely on the number of valid configurations for entire models, partial configurations and containing a specific feature.

For each application, we provide a qualitative explanation and a formal description. We separated the applications depending whether they rely on computing the number of valid configurations **(1)** of an entire feature model, **(2)** containing a specific feature (i.e., commonality), and **(3)** including or excluding sets of features (i.e., partial configurations). The specific algorithms to compute results for these three analyses are discussed in Chapter 4. During the remainder of this chapter, we use Figure 3.1 as a running example. The feature diagram of the simplified car was already introduced in Chapter 2.

3.1 Number of Valid Configurations

A valid configuration satisfies all constraints given by the feature model *FM* [KZK10]. The number of valid configurations corresponds to the cardinality of the set of configurations that meet this requirement [KZK10]. This is trivial for feature models without cross-tree constraints as there are no interdependencies between features from different sub-trees [HGFACC11]. Thus, the number of valid configurations for each

https://doi.org/10.24355/dbbs.084-202009161329-0

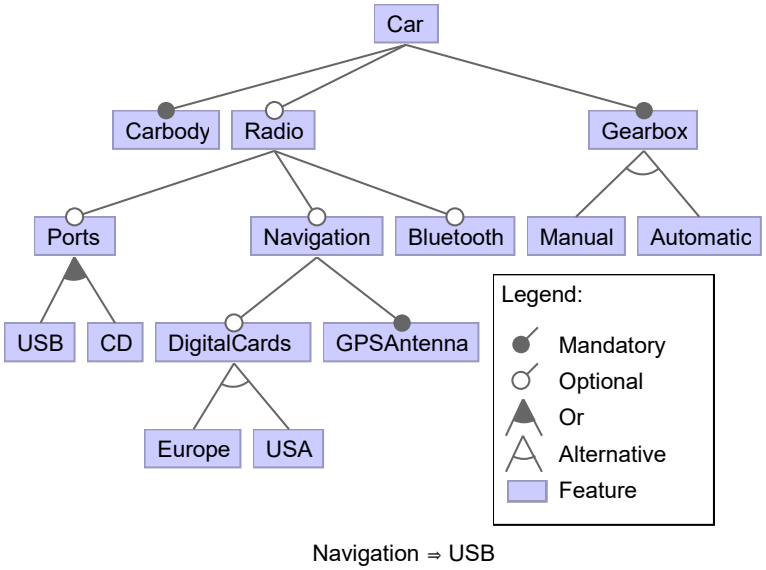

Figure 3.1: Simplified car adapted from Ananieva et al. [Ana16]

sub-tree can be computed independently. The number of valid configurations for the entire feature model can be computed by traversing the feature tree and applying different rules for each relationship-type. For example, the number of valid configurations of an alternative relation is the sum of its children's counts [HGFACC11]. Applying this procedure to our running example while disregarding its cross-tree constraints results in 66 valid configurations. However, with cross-tree constraints this procedure computes wrong results as interdependencies between different feature sub-trees are disregarded. It has been shown that for each propositional formula an equivalent feature model with cross-tree constraints exists [KTM+17]. It follows that an algorithm able to compute the number of valid configurations for any possible feature model is also able to compute the number of satisfying assignments for every propositional formula. Thus, regarding the worst-case complexity, computing the number of valid configurations for a feature model with cross-tree constraints is at least as hard as #SAT which is #P-complete. In the following, we describe eight applications that make use of the number of valid configurations of a feature model.

Void Model

A void feature model FM induces no valid configurations [BSRC10]. Thus, it is not possible to derive even a single configuration from FM which makes the feature model unusable. Computing the number of valid configurations implicitly answers whether a feature model is void. If the returned result $\#FM$ is zero, FM is void. Otherwise, it is not void.

$$\#FM = 0 \iff \text{Void } FM \tag{3.1}$$

The simplified car model induces 42 valid configurations and is, thus, not void. Void models can be identified with satisfiability-based analyses [BSRC10]. We expect that instead computing the number of valid configurations is more expensive. However, if

https://doi.org/10.24355/dbbs.084-202009161329-0

the number of valid configurations is required anyways, the runtime of an additional void analysis can be saved. Analogous, if the model has been found void by a prior analysis, a #SAT call is not required.

Variability Factor

The variability factor of a feature model describes the share of configurations that are valid [BTRC05, FAHCC14, HGFACC11, HFACA13]. The value lies between zero and one. A variability factor close to zero indicates a highly restricted feature model, while a product line with no restrictions has a variability factor of one [HGFACC11, HFACA13]. Disregarding all restrictions, every combination of features induces a valid configurations. In this case, there are $2^{|FEATS_{FM}|}$ configurations. The variability factor of a feature model FM is computed as follows.

$$\text{VariabilityFactor}(FM) = \frac{\#FM}{2^{|FEATS_{FM}|}} \tag{3.2}$$

Benavides et al. [BTRC05] argue that the variability factor can be used to estimate the benefits of a product-line approach. A small variability factor may indicate that developing standalone products is more beneficial. Additionally, an unexpected variability factor may indicate a design error.

The simplified car model 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 fraction of all configurations induced by FM is valid. This shows that the hierarchy of a feature model already limits the set of valid configurations by a large margin. For example, an alternative of ten features induces ten valid configurations. Without the limitation of the alternative, there are $2^{10} = 1024$ configurations.

The already strong limitations of the tree structure may make it difficult to grasp the impact of cross-tree constraints. We propose to simplify this by comparing the number of valid configurations of FM with and without cross-tree constraints. Let FM' be an adaption to FM that does not contain cross-tree constraints.

$$\text{VariabilityFactor}_{CTC}(FM) = \frac{\#FM}{\#FM'} \tag{3.3}$$

Without cross-tree constraints, our running example induces 66 valid configurations. Thus, 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. A $\text{VariabilityFactor}_{CTC}$ close to one also indicates that approximating count based analyses by evaluating the feature model without cross-tree constraints provides more accurate results than a small $\text{VariabilityFactor}_{CTC}$.

Variability Reduction

The number of valid configurations can also be used to examine the impact of a change to the feature model. Consider the feature model FM and a possible adaption FM' of it. We call the change in the number of valid configurations from FM to FM' variability reduction.

https://doi.org/10.24355/dbbs.084-202009161329-0

$$\text{VariabilityReduction}(FM') = \#FM - \#FM' \tag{3.4}$$

$$\text{RelativeVariabilityReduction}(FM') = \frac{\#FM}{\#FM'} \tag{3.5}$$

One use-case of this metric is the identification of undesired changes to the product line caused by the edits. If the number of valid configurations changes in an unexpected way, this is an indicator for a faulty edit. For example, an additional constraint that does not change the number of valid configurations is redundant [KZK10]. Also, new constraints may unexpectedly reduce the number by a large margin. In this case, the edits can be re-evaluated and, thus, design flaws can be prevented. Consider the following change to our example Figure 3.1. A company sells the cars with *Automatic* exclusively to the USA. Thus, the developer wants to introduce a new cross-tree constraint $Automatic \Rightarrow USA$. 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 only affects cars with *Automatic*, we know that there are only 4 valid configurations with *Automatic* left. This information can be used by the developer to reconsider the change to the feature model. A possible alternative may be $DigitalCards \wedge Automatic \Rightarrow USA$. 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. Thus, such design flaws are difficult to detect using traditional satisfiability-based analyses.

Reducing the variability of a product line may simplify quality assurance and maintenance [KZK10], as fewer products need to be considered. In order to do so effectively, it is necessary to know the impact of an edit regarding the reduction of variability. However, it is difficult or even impossible to estimate the difference in the number of configurations without automated support. The number of valid configurations 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 motivating example: removing the gearbox type *Manual* or make *Radio* mandatory. These result in 21 or 40 remaining valid configurations, respectively. Thus, removing *Manual* is more beneficial only considering the variability reduction.

Other works considered computing the added and removed configurations explicitly [TBK09, AHC+12]. While this provides the variability reduction implicitly and additional useful information, 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 [STS20]. 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 in this case but we can compute the number of added and removed configurations.

Rating Errors

Kübler et al. [KZK10] propose using the variability of a feature model specialization for rating an error. A specialization imposes additional constraints to the feature

https://doi.org/10.24355/dbbs.084-202009161329-0

model which removes valid configurations from the configuration space. 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.

$$\#FM_{ERR_1} > \#FM_{ERR_2} \Rightarrow FM_{ERR_1} \text{ is potentially more critical than } FM_{ERR_2} \tag{3.6}$$

Suppose there are two feature subsets that cause errors in our car product line: {*Bluetooth*, *USB*} and {*Automatic*, *GPSAntenna*}. Those subsets appear in 16 and 12 valid configurations, respectively. This can be used as an indicator to argue that the error in {*Bluetooth*, *USB*} is more critical and that the developers should focus on this error first.

Degree of Orthogonality

Feature-model analyses are typically difficult because of the interdependencies between different sub-trees specified by cross-tree constraints. If the impact of these interdependencies is small, the sub-trees can be evaluated separately to get an estimated result. To analyze a single sub-tree, it is possible to use only local constraints [CK05b, BSRC10]. Czarnecki et al. [CK05b] and Benavides et al. [BSRC10] define local constraints as constraints that are imposed by the tree-structure and cross-tree constraints that only contain a single feature. For more precise estimations, we propose to also consider cross-tree constraints that only contain features of the same sub-tree.

The degree of orthogonality describes the ratio between the number of valid configurations regarding all and only local constraints [CW07]. A high degree of orthogonality indicates that interdependencies between subtrees have a low impact regarding the number of valid configurations. In this case, analyses (e.g., the decision about feature selections) can be performed locally [CW07]. 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.

$$\text{DegreeOfOrthogonality}(FM) = \frac{\#FM}{\#FM_{local}} \tag{3.7}$$

For our example, the feature model induces 66 valid configurations if we only consider local constraints (i.e., disregard the cross-tree constraint in our example). Thus, the $\text{DegreeOfOrthogonality}(FM) = \frac{42}{66} \approx 0.636$ indicates that analyses that only consider local constraints may provide a reasonable estimate depending on the accuracy required for the application.

Configuration Relevance

We propose using the number of valid configurations to rate a configurations relevancy. For example, consider 1000 products that were ordered from a product line. One product based on configuration C_1 was ordered 20 times. Without knowing

https://doi.org/10.24355/dbbs.084-202009161329-0

the size of the configuration space, it is difficult to tell whether the configuration is more or less represented than an arbitrary configuration. The product line may contain 42 valid configurations like our motivating example or maybe 10^{11} like a smaller industrial model we evaluated in previous work [STS20]. In the first case, the configuration has a below-average representation but in the second a highly above-average representation. Let S be the sample of configurations, C_1 the configuration in question, and $|S_C|$ the number of occurrences of C in the sample.

$$\text{RelevanceOfConfiguration}(FM, S, C_1) = \frac{|S_{C_1}|}{|S|} * \#FM$$

$\text{RelevanceOfConfiguration}(FM, S, C_1) = a$ states that C_1 appears in a times as often in S compared to an arbitrary configuration. A relevancy score a between zero and 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 PL

Several authors use the number of valid configurations to evaluate whether it is beneficial to develop a product family as product line (PL) or not (i.e., as standalone products) (NPL) [CMC05, HFACA13]. The costs $CNPL$ to develop standalone products can be estimated by multiplying the number of valid configurations $\#FM$ with an cost estimation CP to develop a single product.

$$CNPL(FM) = \#FM \times CP \tag{3.8}$$

For the cost of a product line approach, the authors consider first building one standalone product [CMC05, HFACA13]. 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 [HFACA13]. The details on how to compute CPR are omitted in the following formula as they are not related to counting the number of valid configurations.

$$CPL(FM) = CR \times CP + (\#FM - 1) \times CPR \tag{3.9}$$

Overall, we are interested in the difference between the costs of implementing the family as product line and costs of all standalone products. This difference indicates the benefit of a product-line approach [CMC05, HFACA13].

$$CSAVE(FM) = CNPL(FM) - CPL(FM) \tag{3.10}$$

If the result $CSAVE$ is positive, it is beneficial to build a product line. Both CPL and $CNPL$ are dependent on knowing the number of valid configurations

https://doi.org/10.24355/dbbs.084-202009161329-0

$\#FM$ [CMC05, HFACA13]. Consider the following example: Suppose the cost of developing a standalone car of our example is $CP = 3$ and the cost for developing for reuse is instead 6 which means that $CR = 6$. Furthermore, the cost for developing a new product afterwards with the product line approach is $CPR = 2$. Overall, this results in $CNPL(FM) = 42 * 3 = 126$ and $CPL(FM) = 2 * 3 + 41 * 2 = 87$. $CSAVE(FM) = 126 - 87 = 39$ indicates that it requires less effort to build all 42 possible cars with a product-line approach.

Variability of Feature Sets

Analyzing large feature models is expensive. In some instances, it may be sufficient to focus analyses on specific parts of the feature model. We propose using the number of valid configurations to compute the variability of only a subset of features (e.g., a feature model sub-tree). This can be used to compute the number of valid configurations disregarding *abstract* features. Abstract features are irrelevant for the actual products and are used to structure the feature model [TKES11]. The variability of a feature subset can also be used to find parts that have particularly high or low variability. The identified parts can be used to set a focus for other analyses. Furthermore, an unexpected degree of variability for a subset may indicate modeling errors.

Given a set of features $S \subseteq FEATS_{FM}$, the idea is to compute the number of variants induced by the different combinations of S. For each valid configuration C, every feature $f \notin S$ is discarded. Then, configurations C_i, C_j that include (i.e., $I_{C_i} = I_{C_j}$) and exclude (i.e., $E_{C_i} = E_{C_j}$) the same features are merged. The result is a set D_S of distinct configurations that only contain features included in S. We are interested in the cardinality of this set. The absolute cardinality or in relation to $\#FM$ may be interesting.

$$\text{AbsoluteVariability}(S) = |D_S| \tag{3.11}$$

$$\text{RelativeVariability}(S) = \frac{|D_S|}{\#FM} \tag{3.12}$$

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

To set the variablity of a feature set into perspective, we propose comparing it to the average variabilities of multiple user defined feature sets. Suppose, the feature model is separated in n distinct feature sets whose variability should be compared. Let $\#S_i$ be the variability of feature set S_i. On average each part S has a variability of $\frac{\sum_{i=1}^{n} \#S_i}{n}$. Thus, $\text{AbsoluteVariability}(S) > \frac{\sum_{i=1}^{n} \#S_i}{n}$ shows that S has a higher variability than the average of the other $n - 1$ parts.

https://doi.org/10.24355/dbbs.084-202009161329-0

$$\text{AbsoluteVariability}(S) > \frac{\sum_{i=1}^{n} \#S_i}{n} \Rightarrow S \text{ has an above-average variability.} \quad (3.13)$$

Consider the above mentioned example and the two following feature sets: the subtree induced by *Radio* (S_{Radio}) and the rest of the feature model (S_{Rest}). Thus, n is two, $\#S_{Radio} = 21$, and $\#S_{Rest} = 2$. AbsoluteVariability(S_{Radio}) $= 21 > \frac{21+2}{2} = 11.5$ indicates that S_{Radio} has a high variability.

3.2 Commonality

The commonality of a feature is the relative share of valid configurations that include that feature [FAHCC14, PHRC06, TBC06]. Consequently, given an arbitrary valid configuration C, the commonality of the feature f describes the probability of f appearing in C [PHRC06]. Let FM_f be a specialization of the feature model FM that always includes the feature f (i.e., f is a core feature in FM_f). The commonality of feature f is then described by the following formula [FAHCC14, PHRC06, KZK10].

$$\text{Commonality}(FM, f) = \begin{Bmatrix} \frac{\#FM_f}{\#FM} & \#FM \neq 0 \\ 0 & \#FM = 0 \end{Bmatrix} \quad (3.14)$$

If the feature model is void, $\frac{\#FM_f}{\#FM}$ is undefined. In this case, we consider the commonality of the feature to be zero. Sometimes, commonality also refers to the absolute number $\#FM_f$ of valid configurations containing the feature [BSTC07, CE11]. We use both types, relative and absolute commonality in the remainder of this thesis. If not stated otherwise, we refer to the relative commonality. In the following, we describe eight use cases for computing the commonality of features.

Dead & Core & False-Optional Features

The commonality of a feature can be used to identify defects. Given a feature model FM, a feature f, and the commonalities of $P(f), f$ 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.

$$\text{Commonality}(FM, f) = 1 \iff f \text{ is a core feature [PM16]} \quad (3.15)$$

$$\text{Commonality}(FM, f) = 0 \iff f \text{ is a dead feature [TBC06, PM16]} \quad (3.16)$$

$$\begin{aligned} \text{Commonality}(FM, P(f)) = \text{Commonality}(FM, f) \text{ and } f \text{ is not mandatory} \\ \iff f \text{ is false-optional} \end{aligned} \quad (3.17)$$

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 for other analyses, these defects can be identified without further computations.

https://doi.org/10.24355/dbbs.084-202009161329-0

Atomic Set Candidates

When the commonality of features is computed anyway, the information can be used 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. The following formulas can be used to find atomic set candidates only using the absolute commonality of the features.

$$\#FM_f = \#FM_g \Rightarrow f, g \text{ are candidates for an atomic set} \tag{3.18}$$

$$\#FM_f \neq \#FM_g \Rightarrow f, g \text{ not part of the same atomic set} \tag{3.19}$$

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 commonality can be used as an indicator for the importance of a feature when developing the product line. For example, a developer may have to decide between two alternative features f_1, f_2 to develop next. To potentially create more distinct products, it is beneficial to develop the feature with a higher commonality first. Another example is a supply shortage that for a resource that is required for both f_1 and f_2. Prioritizing the feature with a higher commonality for the allocation of the resource, potentially allows to sell more distinct products.

$$\begin{aligned} \text{Commonality}(FM, f_1) > \text{Commonality}(FM, f_2) \text{ and prioritize } f_1 \text{ over } f_2 \\ \Rightarrow \text{ Build more distinct products} \end{aligned} \tag{3.20}$$

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

In testing, it may be also interesting to prioritize features with a low commonality as features with a high commonality 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. The knowledge about the commonality of features 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.

https://doi.org/10.24355/dbbs.084-202009161329-0

Heavily Constrained Features

We argue that commonality 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' of it that contains no cross-tree constraints. The tree hierarchy of both models FM and FM' is equal. Then, we compare the absolute commonalities of a feature f in FM and in FM'. This information can be used to identify design errors.

$$\text{CTCRestrictiveness}(f) = 1 - \frac{\#FM_f}{\#FM'_f} \tag{3.21}$$

The CTCRestrictiveness lies between zero and one. Zero indicates that the variable is not restricted at all through cross-tree constraints. A value close to one indicates that the feature is heavily restricted through cross-tree constraints. Due to modernization of automatic cars, *CD* is only available for *Manual* cars. The developer introduces a new constraint $CD \iff Manual$ which unintentionally also requires every manual car to have a *CD*. These changes result in a CTCRestrictiveness$(Manual) = 1 - \frac{10}{66} = 0.85$ which indicates that *Manual* is heavily restricted. This information may alert the developer to reconsider the added cross-tree constraint and change it to $CD \implies Manual$ instead.

Payoff Threshold of a Feature

Typically, it is more expensive to develop a feature such that it can be used for multiple products instead of one. However, the required adaptions to use it in other products is then 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 absolute commonality of a feature may indicate whether it is beneficial to develop the feature for reuse.

In general, if developing a feature for reuse is more expensive, a feature needs to be reused a certain number of times before the development for reuse yields a benefit. This break-even threshold is dependent 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 f is smaller than the threshold, it is considered to be efficient to develop the feature for reuse [HGFACC11, HFACA13]. Let $\text{Cost}_f = 1$ be the relative cost of implementing the feature f from scratch. $\text{CostToDevelopForReuse}_f = a$ 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 the feature for a specific product only needs a proportion of the original cost. This reduced cost is described by $CostToReuse_f$.

$$\text{PayoffThreshold}(f) = \frac{\text{CostToDevelopForReuse}_f}{1 - \text{CostToReuse}_f} \tag{3.22}$$

$$\#FM_f < \text{PayoffThreshold}_f \Rightarrow \text{ It is not efficient to develop } f \text{ for reuse.} \tag{3.23}$$

https://doi.org/10.24355/dbbs.084-202009161329-0

In our motivating example, consider the feature *USA*. Suppose the cost of developing the feature for reuse is doubled. Furthermore, reusing *USA* costs $\frac{2}{3}$ of the regular costs of implementing it. This results in threshold of $\frac{2}{1-\frac{2}{3}} = 7$. Thus, *USA* needs to be reused at least seven times. Otherwise, developing the feature for reuse is cost inefficient. *USA* appears in eight valid configurations. As the threshold of for the number of reuses is seven, this indicates that it is beneficial to develop *USA* for reuse.

Several authors proposed concrete procedures to compute the CostToReuse_f and $\text{CostToDevelopForReuse}_f$ [BBMY04, NdAM08]. However, the exact computations are beyond the scope of this thesis as we focus on the dependency of the threshold on computing the commonality. In both publications, the authors considered product lines with a single digit number of products to compute payoff thresholds. We expect that the analysis only provides limited insights for industrial feature models as they induce a large number of valid configurations, typically more than 10^{10} [KZK10, STS20].

Degree of Reuse

When implementing a product line, the degree of reuse can indicate 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 is 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 [HFACA13, FAHCC14, Coh03]. The following formula quantifies the degree of reuse for the feature model FM with the set of common features $COMMON_{FM} \subseteq FEATS_{FM}$ [HFACA13, FAHCC14]. $Cost_f$ is supposed to indicate the effort to develop the feature f.

$$\text{DegreeOfReuse}(FM, COMMON_{FM}) = \frac{\sum_{f \in COMMON_{FM}} (Cost_f * \#FM_f)}{\sum_{f \in FEATS_{FM}} (Cost_f * \#FM_f)} \tag{3.24}$$

It is important to note that industrial feature models typically contain no features that appear in exactly one valid configuration. 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 are three additional cars of the product family described by our running example Figure 3.1 that each contain a unique feature that is not part of the feature model. Every feature that directly appears in the 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 2. This leads to: $\text{DegreeOfReuse} = \frac{15}{15+6} = \frac{15}{21} = 0.71$. The result shows that the main effort to build the products comes from common feature. This indicates a great benefit of the product line approach.

Homogeneity

The homogeneity of a product line describes the similarity of the valid configurations [FAHCC14, CMC05, HFACA13, HGFACC11]. The homogeneity is a value

https://doi.org/10.24355/dbbs.084-202009161329-0

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 of one indicates that every configuration is the same (i.e., there is only one) [FAHCC14]. Clements et al. [CMC05] proposed to compute the homogeneity of a product line using the number of features that appear only in one valid configuration with the following formula. Let $FEATS_{unique} \subseteq FEATS_{FM}$ be the set of features that only appear in one valid configuration.

$$Homogeneity_{Clements}(FM) = 1 - \frac{|FEATS_{unique}|}{|FEATS_{FM}|} \tag{3.25}$$

Fernandez et al. [FAHCC14] argued that this formula computes unexpected results for some cases. Consider a product line that contains 100 features and induces 100 valid configurations with each feature appearing in exactly two different configurations. For this product line, we expect a low homogeneity. However, Equation 3.25 computes $1-\frac{0}{100} = 1$ which indicates that every configuration is the same. Therefore, the authors proposed using the commonality mean of all features [FAHCC14].

$$Homogeneity(FM) = \frac{\sum_{f \in FEATS_{FM}} Commonality_f(FM)}{|FEATS_{FM}|} \tag{3.26}$$

For the above mentioned example, Equation 3.26 results in $Homogeneity(FM) = \frac{\sum_{i=1}^{100} \frac{2}{100}}{100} = \frac{\frac{2*100}{100}}{100} = \frac{2}{100}$, which indicates a low homogeneity of the product line. The low homogeneity of the product line matches the expectation. The homogeneity of our car example is 63% which indicates that the configurations of the product line are relatively similar.

Configuration Derivation Optimizations

Deriving a configuration can also be improved with tool support. One popular example is selection propagation [SSK+20]. Another option is ordering the features in with different strategies to accelerate the derivation. First, a different feature order may reduce the number of configuration steps. Consider an alternative relation with a high number of features. Additionally, each of these features require other features outside of the alternative relation to be selected or deselected. After selecting one of the features, the selection is propagated. Thus, selecting a feature of the group early in the configuration process reduces the number of necessary steps. 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. [MDSD14] proposed six different heuristics to order features to accelerate the derivation. Two of the presented heuristics are dependent on #SAT. The first heuristic orders the features by their commonality. The idea is to process features first that appear in a large variety of products [MDSD14]. 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 commonality reduces the remaining possible

https://doi.org/10.24355/dbbs.084-202009161329-0

selections by a smaller margin. It is important to note that the authors only consider the possibility of selecting a feature and not explicitly deselecting features. Deselecting a feature with a high commonality would reduce the remaining assignment by a larger 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 [MDSD14].

Chen et al. [CE11] also propose to sort the features by their selectivity to optimize the product derivation. The selectivity of a feature is dependent on its commonality and the features that are automatically selected and deselected because of selection propagation. A highly selective feature has a high commonality and selecting it directly implies the selection of a high number of other features. If the selection of a feature propagates to a high number of other features, the number of required configuration steps is reduced. It is important to note that the authors do not consider the possibility of explicitly deselecting a feature. The authors also argue that the commonality of a feature indicates its importance in the product line, as a feature with a high commonality appears in more configurations. The authors also performed an empirical evaluation. The results strongly indicated that selecting highly selective features first accelerates the derivation [CE11]. However, their evaluation does not allow conclusions about the benefits of including commonality for the computation of selectivity. Overall, the benefit of including the commonality of features in the metric is not clear.

3.3 Partial Configurations

Commonality is limited to single features. For some use cases, it is necessary to consider sub-sets of the configuration space that include or exclude multiple features. Given a configuration $C = (FM, I, E)$, $\#FM_C$ refers to the number of valid configurations that include every feature $i \in I$ and exclude every feature $e \in E$. If C is a full configuration, $\#FM_C$ is one. If C is partial, $\#FM_C$ is dependent on the remaining unassigned features $f \notin I \cup E$. In the following, we describe four applications for counting the number of remaining valid configurations for a partial configuration.

Uniform Random Sampling

While some anomalies can be found for an entire product line (e.g., dead feature analysis [SSK+20, BSRC10]), a majority of operations for quality assurance works on specific configurations, such as testing [MKR+16]. Typically, the number of valid configurations is growing exponentially with the number of features for configurable systems [MKR+16]. Thus, analyzing all configurations is often infeasible. One solution to solve this problem is creating a small subset used for analysis. This procedure is called sampling. The literature considers multiple strategies to create efficient samples (e.g., t-wise and random sampling) [MKR+16].

Truly random configurations are useful as they can be used for representative testing, provide accurate statistical data, and can be used to evaluate other sampling methods [OGB+19, CFM+15]. However, it is not trivial to create uniformly distributed

https://doi.org/10.24355/dbbs.084-202009161329-0

random configurations. Randomly including or excluding each feature typically results in a high number of invalid configurations [OBMS16]. Even after removing all invalid configurations from a sample created by this procedure, there is no guarantee for every configuration to have the same chance to appear in the sample [OBMS17].

Uniform random sampling generates such samples that guarantee true randomness of the included configurations. The idea is to create a one-to-one mapping between integers $r \in [1, \#FM]$ and the valid configurations of FM [OBMS17, MOP+19]. 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 [OBMS17, MOP+19]. We discuss actual algorithms to perform uniform random sampling in Chapter 4.

Atomic Sets

In Section 3.2, we describe how to use commonality to find candidates for atomic sets. This procedure can be extended 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 set 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 $V \subseteq FEATS_{FM}$ be a set of features. The number of valid configurations that contain V is equivalent to $\#FM_{C_V}$ with $C_V = (FM, V, \emptyset)$.

$$V \subseteq FEATS_{FM}, v_i \in V : \#FM_{f_1} = \#FM_{f_2} = \cdots = \#FM_{f_n} = \#FM_{C_V} \\ \Rightarrow \text{V is atomic set.} \tag{3.27}$$

In our running example, *Navigation* and *GPSAntenna* appear in 24 valid configurations. Thus, they are potential candidates for an atomic set as stated in the previous section about finding candidates for atomic sets6. Computing $\#FM_{C_V}$ with $V = \{Navigation, Navigation\}$ also results in $\#FM_{C_V} = 24$ valid configurations. Thus, *Navigation* and *GPSAntenna* are part of the same atomic set. *Manual* and *Automatic* both are part of 21 valid configurations. However, computing $\#FM_{C_V}$ with $V = \{Manual, Automatic\}$ results in zero valid configurations. Therefore, the features are not part of the same atomic set.

Rate Feature Interactions for Sampling

We propose to use number of valid configurations that contain a feature set to rate the relevancy of feature interactions. Hereby, interactions that appear in a low or a high number of valid configurations may be interesting depending on the application. For example, the number of valid configurations that contain a feature interaction can be used as a metric for sampling to rate configurations. Krieter et al. [KTS+20] proposed 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 interesting to add

https://doi.org/10.24355/dbbs.084-202009161329-0

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 configuration induced by the feature model. 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.

$$\#FM_{i_1} > \#FM_{i_2} \Rightarrow score(i_1) < score(i_2) \tag{3.28}$$

In our motivating 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. A configuration that contains the latter interaction should have a higher score, as it is more likely to hit a configuration containing *Radio* and *Manual* in following iterations.

Interactive Support for Derivation of Configurations

Deriving valid configurations for a feature model is complex because of all the constraints imposed by the model. It is typically impossible for users to grasp the impact of a 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 selecting it 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 of the product line.

3.4 Summary

In this chapter, we presented 20 applications dependent on counting valid configurations of product lines to motivate the usage of #SAT solvers for analyses in the product-line domain. The described applications can be used for economic estimations, detecting design flaws, simplifying other analyses, extracting statistical information for further use, testing, or interactive support of users and developers. Each application is based on one of the following analyses: **(1)** compute the number of valid configurations for a feature model, **(2)** compute the commonality of features, and **(3)** compute the number of valid configurations of a partial configuration. In the next chapter, we describe algorithms and optimizations for each of those analyses.

https://doi.org/10.24355/dbbs.084-202009161329-0

https://doi.org/10.24355/dbbs.084-202009161329-0

4. Algorithms & Optimizations

In this chapter, we present algorithms to compute the applications described in Chapter 3. The algorithms consist of adaptations from the literature and our own proposals. Furthermore, we provide possible optimizations for the different algorithms. This chapter is especially relevant to readers that want to implement the previously presented applications on their own. Furthermore, the algorithms build the basis for the implementation shown in Chapter 5 and the empirical evaluation presented in Chapter 6.

We provide a variety of algorithms and optimizations that aim to reduce the number or the required runtime of #SAT calls. These either invoke #SAT solvers or employ knowledge compilation to BDD or d-DNNF. Similar to BDDs, d-DNNFs require one expensive offline translation from the original formula to d-DNNF. Afterwards, the number of satisfying assignments can be computed in polynomial time [Dar01a]. The benefit of d-DNNFs is that they are by definition more succinct than BDDs [DM02]. Our previous results also indicate that they scale significantly better for our use case of counting the number of valid configurations of a feature model [STS20]. We propose using d-DNNFs also for commonality, counting remaining valid configurations of a partial configuration, and uniform random sampling. As an example, performing uniform random sampling with a regular #SAT solver requires up to $|FEATS_{FM}|$ #SAT calls for a single configuration [OGB+19]. The d-DNNF only needs to be computed once to create an arbitrary number of configurations. Given the d-DNNF, the queries to compute the single configurations have a polynomial time complexity [DM02].

Most applications presented in Chapter 3 are usages of three analyses, namely number of valid configurations of a feature model, commonality of features, and remaining valid configurations of partial configurations. Thus, the provided algorithms and optimizations focus on these three metrics (in Section 4.1, Section 4.2, and Section 4.3, respectively). For each metric, we provide a base algorithm that is supposed to explain the procedure in a simple way. After each base algorithm, we discuss possible optimizations and alternatives to it. While uniform random sampling is heavily dependent on partial configurations, we decided to separate it in an

https://doi.org/10.24355/dbbs.084-202009161329-0

own section (Section 4.4). This is due to its relevancy and several optimizations that are specific for uniform random sampling.

4.1 Number of Valid Configurations

The number of valid configurations of a feature model can be computed using a #SAT solver. In order to do so, we translate the feature model to an equivalent propositional formula, typically in CNF, and give it as input to the #SAT solver. The number of satisfying assignments is then equal to the number of valid configurations. Algorithm 5 describes this procedure.

Algorithm 5 `NumberOfValidConfigurations`(FM)

1: CNF_{FM} := `convertToCNF`(FM)
2: $\#CNF_{FM}$:= `execute#SAT`(CNF_{FM})
3: **return** $\#CNF_{FM}$

Optimizations

In this section, we describe procedures that aim to decrease the time required to compute the number of valid configurations. The optimizations considered can also be used to improve other analyses (e.g., computing commonality).

Simplify Feature Model Formula

We propose to use domain-specific formula pre-processing to reduce the runtime required for the #SAT calls. Properties that are available due to previous feature-model analyses can be used to simplify formulas. We consider using knowledge about core features, dead features, and redundant constraints for the simplification. Literals corresponding to a core feature have to be assigned true. Otherwise, the formula is not satisfiable. Thus, each core feature can be added as a unit clause. We expect that adding unit clauses increases the speed of the computation as modern #SAT solvers [Thu06, BJP00, SBB+04, Bie08] and d-DNNF compilers [LM17, MMBH10, Dar02] typically use unit propagation. Alternatively, one may update the entire CNF by replacing core features with $\top$ and dead features with $\bot$. Constraints can then be updated or entirely removed using the rules shown in Equation 4.1 and Equation 4.2 [Tiu98]. Let c be a variable corresponding to a core feature and F an arbitrary propositional formula. $\top$ and $\bot$ correspond to true and false, respectively as introduced in Chapter 2.

$$\begin{aligned} c \wedge F &\equiv F \\ c \vee F &\equiv \top \\ \top \wedge F &\equiv F \\ \top \vee F &\equiv \top \end{aligned} \tag{4.1}$$

After propagating the changes resulting from a core feature to a cross-tree constraint, the cross-tree constraint may become a tautology. A cross-tree constraint that is

https://doi.org/10.24355/dbbs.084-202009161329-0

equivalent to $\top$ can be omitted. Let d be a variable corresponding to a dead feature and F an arbitrary propositional formula.

$$\begin{aligned} d \wedge F &\equiv \bot \\ d \vee F &\equiv F \\ \bot \wedge F &\equiv \bot \\ \bot \vee F &\equiv F \end{aligned} \tag{4.2}$$

A cross-tree constraint that is equivalent to $\bot$ causes the feature model to be void. A redundant constraint has no impact on the set of valid configurations and can be just removed. However, the removal of multiple redundant constraint at the same time may change the semantics of a feature model. For example, two constraints ct_1 and ct_2 with $ct_1 \equiv ct_2$ may both marked redundant. If ct_1 is removed, ct_2 may not be redundant anymore. Thus, after the removal of ct_1 the status of ct_2 has to be reviewed with a SAT solver once again. Suppose a constraint is removed and no other changes are made to the feature model. In this case, it is not possible for a constraint that was not redundant before the change to be redundant afterwards. Thus, the list of redundant constraints from the current iteration can be used as a list of potential candidates for remaining redundant constraints of the following iteration after deleting a redundant constraint. However, it is unclear whether the effort required to compute redundant constraints is worth given that only potentially a few constraints are removed.

CNF Translation Techniques

CNF is a common format for propositional formulas when using SAT or #SAT solvers [OGB+19, Thu06, BSB15, Bie08, MMZ+01]. Even though every propositional formula F can be translated to CNF using logic equivalence rules (e.g. De Morgan rules), this often results in large CNFs [OGB+19]. Thus, other procedures have been proposed that create an equisatisfiable CNF. However, some of those are not applicable for #SAT as it is possible that an equisatisfiable translation changes the number of solutions if new variables are introduced [OGB+19]. Introducing new variables does not necessarily increase the number of satisfying assignments [OGB+19]. For example, Tseytin's transformation introduces new variables but no additional satisfying assignments [OGB+19, Tse83]. If no new variables are introduced, an equisatisfiable translation does not change the number of satisfying assignments [OGB+19]. The performance of SAT solvers can be improved by using another equivalent CNF [NW01, OGB+19]. Thus, one way to decrease the runtime of the #SAT calls is finding an effective translation technique that is applicable for #SAT (i.e., does not change the number of solutions).

Variable Ordering

DPLL is internally used by a majority of #SAT techniques [BJP00, SBK05a, Thu06, BSB15, Bie08, Dar04, LM17, MMBH10, HD04]. The complexity of DPLL for SAT problem instances is, inter alia, dependent on the ordering of variables [HD03,

https://doi.org/10.24355/dbbs.084-202009161329-0

WJHS04, AMS01, DK05]. Thus, changing the order in which variables are processed may reduce the runtime of #SAT calls. First, we present ordering strategies that are applicable for any propositional formula. Second, we provide strategies that exploit the structure of feature models.

Huang et al. [HD03] proposed a variable ordering that enables the decomposition of a formula after as few assignments as possible. A formula that is not decomposable may be decomposable after assigning certain variables. The problem can be split into sub-problems after such assignments and the DPLL algorithm can be accelerated. Thus, assigning these variables early is beneficial. This logic can also be recursively applied to the resulting sub-problems. Another ordering strategy is cube-and-conquer which prioritizes variables whose assignment immediately eliminates a high number of clauses [OGB+19]. An aspect that is used by many heuristics is the activity of a variable (i.e., the number of clauses the variable appears in) [SS03, DK05]. Assigning variables with a high activity potentially resolves more clauses after multiple assignments compared to cube-and-conquer.

The described orderings can also be used to optimize the SAT analysis of feature models. However, it may also be beneficial to exploit the properties of a feature model and additional domain knowledge to order variables [MWCC08]. On the one hand, it may be less complex to find an effective ordering for feature models. On the other hand, an ordering specifically optimized for feature models may improve the performance even more. Thus, we propose two heuristics that exploit properties of a feature model.

Considering a heuristic that aims to decompose the propositional formula with few assignments, it may be beneficial to assign features that are connected to features from other sub-trees via cross-tree constraints. If every feature of a sub-tree that is part of cross-tree constraint with features from other sub-trees is assigned, the sub-tree can be evaluated as an independent sub-problem. In our previously introduced example feature model, if *Navigation* is assigned first, the feature trees induced by *Navigation* and *Ports* can be evaluated separately.

A large alternative induces a high number of clauses that only contains the features appearing in the alternative. Thus, prioritizing features of a large alternative should resolve a high number of clauses which simplifies the following computations. Furthermore, a tool can identify alternative features within linear time in the number of features. Using easily accessible domain knowledge about the feature tree provides potentially beneficial variable orderings efficiently.

Mendonca et al. [MWCC08] propose feature model specific variable orderings to reduce the sizes of BDDs. These may also be beneficial for #SAT. First, the authors propose to order the variables such that children features are as close as possible to their parents. Second, they propose grouping subtrees connected by cross-tree constraints.

DPLL with Alternative Propagation

Typically, parsing the feature tree to propositional logic creates a high number of clauses. This is especially the case for alternatives, as an alternative with n features

https://doi.org/10.24355/dbbs.084-202009161329-0

produces $O(n^2)$ clauses or introduces additional variables [KK07, BTS19]. Furthermore, our data indicates that large alternative groups are prevalent in real-world feature models [STS20]. Thus, we expect a decreased runtime by handling alternatives separated from the propositional formula while performing DPLL. Let a be a feature in an alternative group. While performing a DPLL procedure, assigning true to a could easily propagate false to every other variable in the alternative group. Algorithm 6 describes the procedure which handles the alternative groups separately as seen in lines **8-12**. The remainder of the algorithm is equivalent to the basic counting DPLL Algorithm 2. If the propagation of alternative groups is integrated into the DPLL procedure, clauses that result from translating alternative groups can be omitted. Thus, we expect that the described integration reduces the size of CNFs representing feature models and the time required for boolean propagation of features in an alternative group. However, the technique requires to adapt the underlying solver and, thus, is not easily applicable for off-the-shelf #SAT solvers. We argue that the three optimizations described above, namely simplification of the formula, variable ordering, and CNF translation techniques can be applied to Algorithm 6 to reduce its runtime.

Algorithm 6 `AlternativeDPLL`(F,α)

```
1: if α(F) = ⊤ then
2:     return ⊤
3: end if
4: if α(F) = ⊥ then
5:     return ⊥
6: end if
7: l_next := getNextUnassignedVariable()
8: if isAlternative(l_next) then
9:     α_next := α ∪ {l_next} ∪ getNegatedSiblings(l_next)
10: else
11:    α_next := α ∪ {l_next}
12: end if
13: if AlternativeDPLL(F, α_next) = ⊤ then
14:    return ⊤
15: end if
16: α_¬next := α ∪ {¬l_next}
17: if AlternativeDPLL(F, α_¬next) = ⊤ then
18:    return ⊤
19: end if
20: return ⊥
```

Incremental Encoding During the Evolution of a System

Previous results showed that configuration spaces are continually growing which typically results in continually harder computations during the evolution of a system [STS20]. There are systems whose early evolution steps can be evaluated within seconds but later ones cannot be analyzed even within 24 hours [STS20]. To reduce the runtime of later evolution steps, we aim to reuse results from earlier evolution steps. Our idea is to compute the number of valid configurations of the first

https://doi.org/10.24355/dbbs.084-202009161329-0

evolution step. Then, we compute the following ones by computing the difference. Let FM' be a model evolved from FM and $\#(FM)$ is known. It suffices to know $\delta\#(FM, FM') = \#(FM') - \#(FM)$ to compute $\#(FM')$. We compute $\delta\#$ using the number of added $\#(\neg FM \wedge FM')$ and removed $\#(\neg FM' \wedge FM)$ valid configurations using the following equations Equation 4.3, Equation 4.4, and Equation 4.5. Thüm et al. [TBK09] used this idea to compute the added and removed configurations of feature model edits. For the sake of simplicity, we assume that the sets of features for FM and FM' are equal. Thüm et al. [TBK09] describe how to handle features that appear only in one of the feature models.

$$\begin{aligned}\delta\#(FM, FM') &= \delta_+\#(FM, FM') - \delta_-\#(FM, FM') \\ &= \#(\neg FM \wedge FM') - \#(\neg FM' \wedge FM)\end{aligned} \tag{4.3}$$

$$\#(FM') = \#(FM) + \delta\#(FM, FM') \tag{4.4}$$

$$\#(FM') = \#(FM) + \#(\neg FM \wedge FM') - \#(\neg FM' \wedge FM) \tag{4.5}$$

Algorithm 7 shows the procedure. In lines **2-7**, the number of valid configurations for the first evolution step is computed. Afterwards, the following evolution steps are iteratively analyzed in lines **8-14**. The number of added and removed valid configurations is computed in lines **9** and **10**, respectively. The overall result is computed and stored for the next iteration in lines **11-13**.

Algorithm 7 `Incremental#`(*FmEvolution*)

```
1:  resultList = [ ]
2:  FM_0 = FmEvolution.get(0)
3:  CNF_FM0 = convertToCNF(FM_0)
4:  #FM_0 = execute#SAT(CNF_FM0)
5:  resultList.add(#FM_0)
6:  LastModel = FM_0
7:  LastCount = #FM_0
8:  for FM_i in FmEvolution(1 : n) do
9:      δ_+# = execute#SAT(getCombinedCNF(¬LastModel, FM_i))
10:     δ_-# = execute#SAT(getCombinedCNF(LastModel, ¬FM_i))
11:     #FM_i = LastCount + δ_+# − δ_-#
12:     resultList.add(#FM_i)
13:     LastCount = #FM_i
14: end for
15: return resultList
```

Reusing the number of valid configurations of the previous model FM_i may reduce the time required to compute $\#FM_{i+1}$. Furthermore, Equation 4.5 provides additional insights about the evolution of the configuration space in the number of added and removed valid configurations. If computing one $\delta\#(FM_i, FM_{i+1})$ fails, none of the following evolution steps can be evaluated using the algorithm presented in Algorithm 7. In this case, directly computing $\#FM_{i+1}$ may provide a result.

https://doi.org/10.24355/dbbs.084-202009161329-0

If solver can compute a result, the algorithm can continue as described. Another possibility is to decompose the changes from FM_i to FM_{i+1} into smaller steps. For example, a new version may introduce two new constraints. We first add one of the constraints to FM_i to create an intermediate model FM'. Then, we compute $\delta\#(FM_i, FM')$ and $\delta\#(FM', FM_{i+1})$ to implicitly get $\delta\#(FM_i, FM_{i+1})$. Furthermore, if a later model $FM_j, j > i$ can be evaluated using $\#SAT(CNF_{FM_j})$, the procedure can be performed backwards (i.e., compute FM_{j-1} using $\delta\#(FM_j, FM_{j-1})$. For large changes, it may also be beneficial to skip one step and instead compute $\delta\#(FM_i, FM_{i+2})$. However, we expect that $\delta\#(FM_i, FM_{i+2})$ is more expensive to compute than $\delta\#(FM_i, FM_{i+1})$ for most changes. We expect that the first three optimizations considered for computing the number of valid configurations are applicable without further adaptations for the incremental procedure. DPLL with alternative propagation would require some adaptations as the two feature models that are compared may contain different alternative groups.

Recap

We proposed five optimizations to reduce the required runtime to compute the number of valid configurations of a feature models. Four of those aim to decrease the runtime of the corresponding #SAT call and can be applied to any feature model. The fifth one is only applicable for a set of feature models representing the evolution of the system and potentially reduces the runtime required to analyze multiple models of this set.

4.2 Commonality

For some applications considered in Chapter 3, it is not sufficient to compute the number of all valid configurations. In this section, we discuss algorithms and optimizations for computing the number of valid configuration that contain a certain feature (i.e., the commonality of that feature). To compute the commonalities of features in a feature model using a #SAT solver, we convert the feature model to CNF. For each feature, we conjunct the corresponding literal to the CNF as a unit clause. Each resulting CNF is used as input for a #SAT solver [KZK10]. Algorithm 8 shows this procedure for a single feature.

Algorithm 8 `Commonality`$(FM, f, \#FM)$

1: $CNF_{FM} =$ `convertToCNF`(FM)
2: $CNF_{FM_f} = CNF_{FM} \wedge f$
3: $\#CNF_{FM_f} =$ `execute#SAT`(CNF_{FM_f})
4: **return** $\frac{\#CNF_{FM_f}}{\#FM}$

Optimizations

In the following, we discuss optimizations that are specific for computing the commonality of features. Every optimization we considered for feature models is also

https://doi.org/10.24355/dbbs.084-202009161329-0

applicable for computing commonalities, as the major part of the CNF that represents the constraints imposed by the feature model is equal for both computations. We propose to use the incremental encoding for the evolution of systems for commonality in two ways: (1) the commonality of a feature in a previous version can be used to compute the commonality in the next version and (2) the adapted feature model FM_f for which f is a core feature can be treated as an evolution step following FM.

Result Propagation Using Results from other Analyses

We argue that the commonality of some features can be computed without an extra #SAT call using information resulting from other analyses. In order to do so, we propose to use knowledge about core, dead, and false-optional features to reduce the number of required #SAT calls. First, the commonality of a core feature is always 1. Let $f, g \in FEATS_{FM}$ be features, $p \in FEATS_{FM}$ the parent of f, and A an atomic set.

$$f \text{ is core} \iff \text{Commonality}(FM, f) = 1 \tag{4.6}$$

Computing core features typically requires multiple SAT calls and, thus, is computationally expensive [HPMFA+16]. A subset of the core features can be extracted without SAT calls in linear time in the number of features using the following recursive procedure. First, add the root to the subset. Then, recursively add all mandatory children starting from the root.

The commonality of a dead feature is always zero. Furthermore, the commonality of all features in an atomic set is always equal.

$$f \text{ is dead} \iff \text{Commonality}(FM, f) = 0 \tag{4.7}$$

$$f, g \in A \Rightarrow \text{Commonality}(FM, f) = \text{Commonality}(FM, g) \tag{4.8}$$

Computing atomic sets is an expensive operation [SKT+16]. A mandatory or false-optional child and its parent are always part of the same atomic set. There might be other features in the same atomic set because of cross-tree constraints but parents and their mandatory children can be used to under-approximate atomic sets. These under-approximations can be used reduce the number of #SAT calls without computing every atomic set.

Commonality DPLL

It is also possible to adapt the underlying #SAT procedure to reduce the number of required #SAT calls. Algorithm 8 performs a separate #SAT call for every feature of the feature model. We expect that each computation takes approximately as much time as counting the number of valid configurations. Thus, we propose to adapt the counting DPLL procedure to compute all commonalities within a single run. Each time a partial assignment is found using DPLL, it is trivial to determine how many

https://doi.org/10.24355/dbbs.084-202009161329-0

of the induced satisfying assignments contain a certain variable [SBK05b]. Let F be a formula with n variables and let α be a partial assignment that satisfies F. $n-|\alpha|$ variables are unassigned (i.e., $\alpha(v) = \mathit{UNDEF}$). Thus, α induces $\#\alpha = 2^{n-|\alpha|}$ as every unassigned variable can be either assigned $\top$ or $\bot$ without any impact on the satisfiability of F. $\alpha(v)$ corresponds to the value assigned for v in α. Let $\#\alpha_v$ be the number of satisfying assignments induced by α that contain v. Every v with $\alpha(v) = \top$ appears in each of the $\#\alpha$ assignments. Thus, Equation 4.9 holds. Every v with $\alpha(v) = \bot$ appears in no satisfying assignment. Thus, Equation 4.10 holds. Every v with $\alpha(v) = \mathit{UNDEF}$ appears in half of the satisfying assignments. Thus, Equation 4.11 holds.

$$\alpha(v) = \top \Rightarrow \#\alpha_v = \#\alpha = 2^{n-|\alpha|} \tag{4.9}$$

$$\alpha(v) = \bot \Rightarrow \#\alpha_v = 0 \tag{4.10}$$

$$\alpha(v) = \mathit{UNDEF} \Rightarrow \#\alpha_v = \frac{\#\alpha}{2} = 2^{n-|\alpha|-1} \tag{4.11}$$

These properties can be used to adapt the #SAT DPLL Algorithm 2 presented in Chapter 2 to compute the commonalities of all features. This idea was proposed by Sang et al. [SBK05b] to consider single variables for weighted model counting. The adaptation is shown in Algorithm 9. The sets $\alpha_\top, \alpha_\bot, \alpha_{\mathit{UNDEF}} \subseteq \mathit{vars}(F)$ correspond to the variables that are assigned $\top, \bot, \mathit{UNDEF}$ in α. If a solution is found, the commonalities are computed in lines **1-10**. Equation 4.9 is used to handle variables v with $\alpha(v) = \top$ in lines **2-4**. Equation 4.10 is used to handle variables v with $\alpha(v) = \bot$ in lines **5-7**. Equation 4.11 is used to handle variables v with $\alpha(v) = \mathit{UNDEF}$ in lines **8-10**. If the current formula is unsatisfiable under the current assignment α, a commonality of zero is saved for every feature as seen in lines **11-14**. If the formula is neither already satisfied nor unsatisfiable, the algorithm is recursively called with the next variable assigned to $\top$ and $\bot$ respectively in lines **16-18**. Then, the commonalities for both cases are added for each feature and stored in lines **19-21**.

The procedure skips redundant effort as the DPLL procedure needs to identify the satisfying assignments only once instead of once for every feature. Overall, only one invocation of a #SAT solver is required. However, the procedure requires to adapt the underlying solver and, thus, is not easily applicable for off-the-shelf #SAT solvers. We argue that any optimization considered to reduce the runtime for a #SAT call considered so far is applicable for our commonality adaptation of DPLL as the behavior is analogous to counting DPLL for the most part.

Compute Solution Sets from CTCs

Fernandez et al. [FAHCC14] propose a procedure that aims to reduce the effort required to analyze the feature tree by evaluating it separately. The authors compute disjunct solution sets that are valid only considering the cross-tree constraints

https://doi.org/10.24355/dbbs.084-202009161329-0

Algorithm 9 `CommonalityDPLL`(F, α)

```
1: if α(F) = ⊤ then
2:    for v ∈ α_⊤ do
3:       commonalities.put(v, 2^{n−|α|})
4:    end for
5:    for v ∈ α_⊥ do
6:       commonalities.put(v, 0)
7:    end for
8:    for v ∈ α_UNDEF do
9:       commonalities.put(v, 2^{n−|α|−1})
10:   end for
11: else if α(F) = ⊥ then
12:   for v ∈ vars(F) do
13:      commonalities.put(v, 0)
14:   end for
15: else
16:   l_next := getNextUnassignedVariable()
17:   commonalities_next := CommonalityDPLL(F, α ∪ l_next)
18:   commonalities_¬next := CommonalityDPLL(F, α ∪ ¬l_next)
19:   for v ∈ vars(F) do
20:      commonalities.put(v, commonalities_next.get(v)                +
         commonalities_¬next.get(v))
21:   end for
22: end if
23: return commonalities
```

https://doi.org/10.24355/dbbs.084-202009161329-0

but not the tree-hierarchy. For example, the formula $(A \wedge B) \vee (A \wedge C)$ induces the following satisfying full assignments $\{A, B, \neg C\}$, $\{A, B, C\}$, $\{A, \neg B, C\}$. The assignments $\{A, B, \neg C\}$ and $\{A, B, C\}$ can be merged into the partial assignment $\{A, B\}$ which induces those two solutions. This results in the two solutions sets $\{A, B\}$ and $\{A, \neg B, C\}$. After computing the solution sets, the feature tree is traversed for each set resulting in a number of solutions that results from the induced sub-configuration space. The results can just be summed up as the solution sets are disjunct. Their empirical evaluation indicates that this procedure performs especially well on feature models with a low number of constraints [FAHCC14]. This makes sense as the procedure requires to enumerate all distinct partial solutions resulting from the cross-tree constraints and store them temporarily.

Compute Commonalites with d-DNNF Traversal

We propose a procedure that computes all commonalities within a single traversal of a d-DNNF. Instead of $O(n)$ #SAT calls, the procedure translates the CNF into d-DNNF and traverses the resulting d-DNNF once to compute all commonalities. The algorithm works similarly to computing the overall number of satisfying assignments with d-DNNF as described in Chapter 2. In addition to the current model count, the current commonalities of all features are stored during the traverse. In the following, we refer to the commonality of variable v in the sub-tree induced by the node N as $\#N_v$. $\#N$ refers to the number of satisfying assignments in the sub-tree induced by the node N.

A positive literal corresponding the variable v induces one solutions that contains v (i.e., $\{v\}$). A negative literal induces one solution that does not contain v $\{\neg v\}$ instead. At a positive/negative literal node $L/\neg L$, the current absolute commonality of the variable v which L corresponds to is set to 0/1 as seen in lines **17-30**. At an Or-node (disjunction) O the commonality of v is set to the sum of the child nodes $O_1, ..., O_n$ commonalities $\#O_v = \#O_{1_v} + \#O_{2_v} + ... + \#O_{n_v}$ as seen in lines **10-16**. This works because the children of a deterministic disjunction share no common solutions. Let A_0 be the child of the And-node (conjunction) A that contains the variable v. By definition, the other children of A do not contain v. Thus, the model count of the And-node $\#A_v$ can be computed by multiplying the overall model counts of the children $\#A_v = \#A_{0_v} * \prod_{i=1}^{m} \#A_i$ as seen in lines **2-9**. At the end of the traversal, the commonality of every feature is provided. The procedure is described in Algorithm 10 and Algorithm 11.

Algorithm 10 `DDNNFCommonalities`(Fm)

1: $dDNNF_{FM} =$ `convertToDDNNF`(FM)
2: $ROOT =$ `getRoot`$(dDNNF_{FM})$
3: **return** `getRecursiveCommonalities`$(ROOT)$

The traversal has a computational complexity of $O(n * m)$ for a formula with n variables and m nodes if we assume constant time complexity for arithmetic operations. At each of the m nodes, the current count of all n variables has to be computed. Another observation we made that has not been considered before is that core and dead features can be easily extracted from a smooth d-DNNF without traversing

https://doi.org/10.24355/dbbs.084-202009161329-0

Algorithm 11 `getRecursiveCommonalities`($node$)

1: $commonalities = [\,]$
2: **if** $node$.`isAnd`() **then**
3: **for** $f \in FEATS_{FM}$ **do**
4: $A_0 =$ `getChildContaining`(f)
5: $A_{other} =$ `getChildren`($node$) $\setminus \{c_f\}$
6: $\#A_f = \#A_{0_f} * \prod_{A_i \in A_{other}} \#A_i$
7: $commonalities$.`add`($\#A_f$)
8: **end for**
9: **end if**
10: **if** $node$.`isOr`() **then**
11: **for** $f \in FEATS_{FM}$ **do**
12: $children =$ `getChildren`($node$)
13: $\#O_f = \sum_{O_i \in children} \#O_{i_f}$
14: $commonalities$.`add`($\#O_f$)
15: **end for**
16: **end if**
17: **if** $node$.`isLeaf`() **then**
18: **for** $f \in FEATS_{FM}$ **do**
19: **if** `getVariable`($node$) $\neq f$ **then**
20: $\#L_f = 0$
21: **end if**
22: **if** `isPositive`($node$) **then**
23: $\#L_f = 1$
24: **end if**
25: **if** `isNegative`($node$) **then**
26: $\#L_f = 0$
27: **end if**
28: $commonalities$.`add`($\#L_f$)
29: **end for**
30: **end if**
31: **return** $commonalities$

https://doi.org/10.24355/dbbs.084-202009161329-0

it. A feature is core iff only positive literals corresponding to the feature appear in the smooth d-DNNF. A feature is dead iff only negative literals corresponding to the feature appear in the d-DNNF. We explain how this works inductively for an arbitrary core feature f. In the following, we show that it is not possible to induce a solution that excludes f (i.e., $\alpha(f) = \bot$) without a negative literal $\neg l_f$ that corresponds to f. It is important to note that (1) an And induces no solution with $\alpha(f) = \bot$ if one of its children induces no solution with $\alpha(f) = \bot$ and (2) Or induces no solution with $\alpha(f) = \bot$ if neither of its children does so. A positive literal l_f only induces one solution with $\alpha(f) = \top$. The parent of l_f can either be an And or an Or. For And, it is not possible that the parent induces a solution with $\alpha(f) = \bot$ as its child l_f directly implies $\alpha(f) = \top$ by property (1). For Or, we assume that each of its descendants is either And or a literal without a loss of generality (otherwise we chose one of its descendant $\vee$ which has no $\vee$ as one of its descendants). Due to the smooth property, every child of the chosen Or contains the variable f. We argued that it is not possible to induce a solution that contains f with composition of And and negative literals $\neg l_f$. Thus, neither disjunct of the chosen Or induces a solution with f. (2) implies that the chosen Or induces no solution with f. These properties can be inductively applied to show that the root and, thus, the entire d-DNNF induces no solution with $\alpha(f) = \bot$ (making f a core feature) if there is no negative literal $\neg l_f$ corresponding to f. This works analogously for dead features and positive literals.

We can use this observation to save traversals for all core and dead features, as we can efficiently extract all core and dead features by traversing the leafs of the d-DNNF once. While most optimizations considered thus far are not applicable for the traversal of the d-DNNF, we expect that the optimizations that reduce the time required for a #SAT call (e.g., simplify the feature model formula) can be applied to the translation from CNF to d-DNNF.

Recap

We provided a base algorithm that computes the commonality of each feature with a separate #SAT call. As an optimization for that algorithm, we discussed how to re-uses the results from other analyses to reduce the number of required #SAT calls. In addition, we presented three other algorithms that compute the commonalities by (1) adapting the DPLL procedure, (2) evaluating the feature tree and cross-tree constraints separately, and (3) exploit the properties of a d-DNNF.

4.3 Partial Configurations

The number of remaining valid configurations of a partial configuration can be computed in similar way as commonality. Given a propositional formula CNF_{FM} that represents the feature model FM and a partial configuration $C = (FM, I, E)$, the formula can be adapted in the following way. Each included feature $i \in I$ is conjuncted to CNF_{FM}. For each excluded feature $e \in E$, a negation of the variable corresponding to e is conjuncted to CNF_{FM}. The resulting formula $CNF_C = CNF_{FM} \wedge (\bigwedge_{i \in I} i) \wedge (\bigwedge_{e \in E} \neg e)$ is given as an input to a #SAT solver to compute the number of remaining valid configurations of the partial configuration C.

https://doi.org/10.24355/dbbs.084-202009161329-0

Algorithm 12 #PartialConfiguration($C = (FM, I, E)$)

1: $CNF_{FM} = \texttt{convertToCNF}(FM)$
2: $CNF_C = CNF_{FM} \wedge (\bigwedge_{i \in I} i) \wedge (\bigwedge_{e \in E} \neg e)$
3: $\#CNF_C = \texttt{execute\#SAT}(CNF_C)$
4: **return** $\#CNF_C$

Optimizations

We argue that each of the five considered optimizations for computing the number of valid configurations are applicable for the base algorithm described above. Also, the algorithms and optimization considered for commonality can be adapted for the usage on partial configurations. A partial configuration that excludes a core feature induces zero valid configurations (analogous for included dead features). In this case, a #SAT call is not required for the partial configuration. The solution sets resulting from cross-tree constraints can also be used for partial configurations by applying the solution sets that do not violate the partial configurations (e.g., variable excluded in the partial configuration is assigned $\top$ in the solution set) to the feature tree. In the following, we provide an algorithm that exploits the properties of a d-DNNF to compute the number of remaining configurations of a partial configuration which is an adaptation of the algorithm described for commonality.

Traversing d-DNNF

Computing the remaining valid configurations of multiple partial configurations causes a large redundant effort as the major part of the CNF is equal. With the following optimization, we aim to reduce the effort for multiple sequential queries on partial configurations. The traversal to compute commonalities presented in Section 4.2 can be extended for partial configurations. To this end, the rules for the And-node and Or-node have to be changed as shown in Algorithm 13. Instead of computing the number of satisfying assignments that contain one certain variable as we did for commonality, we compute the number of satisfying assignments that includes a set of variables I and excludes a set of variables E to handle the partial configuration $C = (FM, I, E)$. In the following, we describe the required adaptations to handle partial configurations for the different node types.

For the children of an And-node A, there are two cases to consider as seen in lines **1-6**. First, the child may contain no variable that is included or excluded in the partial configuration C. In this case, the procedure considers the number of all satisfying assignments as there are no additional constraints on the variables imposed by the configuration. Second, the child may contain a variable that is either included or excluded in the partial configuration. In this case, the procedure considers all satisfying assignment that assign $\alpha(i) = \top$ for every $i \in I$ and $\alpha(e) = \bot$ for every $e \in E$. All other satisfying assignments violate the constraints imposed by the partial configuration. These two cases can be used to extract the number of satisfying assignments of each child. The result for the parent And-node is the product of those as explained in Chapter 2. Without a loss of generality, let A_i with $i = 1, ..., n$ be the children of the And-node that contain the subset $S_i \subseteq I \cup E$ of the features included or excluded in the configuration $C = (FM, I, E)$ and A_j with

https://doi.org/10.24355/dbbs.084-202009161329-0

$j = n + 1, ..., m$ be the children that contain no variable $v \in I \cup E$. Furthermore, let $\#A_i$ be the number of satisfying assignments induced by A_i overall and $\#A_{i_S}$ be the number of satisfying assignments that assigned every included variable $i \in S \cap I$ with $\alpha(i) = \top$ and every excluded variable $e \in S \cap E$ with $\alpha(e) = \bot$. Equation 4.12 can be used to compute the number of satisfying assignments with the restrictions of the partial assignment $\#A_C$.

$$\#A_C = (\prod_{i=1}^{n} \#A_{i_{S_i}}) * (\prod_{j=n+1}^{m} \#A_j) \tag{4.12}$$

For this algorithm we limit ourselves to smooth d-DNNFs. Thus, each child of an Or-node O contains the same set of variables $S \subseteq I \cup E$. Therefore, each child of the or node O_i contains the variable sub-set S. It follows that we consider the number of satisfying assignments that assign every included $i \in S \cap I$ with $\alpha(i) = \top$ and every excluded variable $e \in S \cap E$ with $\alpha(e) = \bot$ for every child. The sum of the counts for the children is the result for the parent Or O as the assignments of children are distinct due to the deterministic property. The formula Equation 4.13 is applied in lines **7-11**.

$$\#O_S = O_{1_S} + ... + O_{n_S} \tag{4.13}$$

The number of satisfying assignments for positive and negative literals that correspond to an included feature are set to one and zero, respectively in lines **12-20**. For excluded features, the values are flipped (i.e., zero for positive and one for negative literals) as seen in lines **21-29**. The values for literals are set analogous to d-DNNF-based algorithm for commonality.

Just as for commonality, some queries can be skipped by using knowledge about core and dead features. If a core feature is excluded or a dead feature is included in the partial configuration, the number of valid configurations is always zero. For commonality, we argued that other optimizations can be used to accelerate the compilation from CNF to d-DNNF which should be analogous for partial configurations.

4.4 Uniform Random Sampling

Uniform random sampling creates a sample in which each valid configuration has the same chance to appear in the sample (i.e., the configurations are uniformly distributed). A naive solution may be to enumerate all $\#FM$ valid configurations induced by FM, generate a random number $r \in [1, \#FM]$, and select the r-th configuration. However, this would require to store the actual configurations. This is not feasible for large configuration spaces, as they often contain more than 10^{50} valid configurations [OBMS17, STS20].

Oh et al. [OBMS16, OBMS17] propose using a counting binary decision diagram (CBDD) for uniform random sampling. A CBDD is a special form of BDDs that additionally stores the number of remaining solutions for each edge. This allows a traverse in linear time in the number of features to find the r-th configuration

https://doi.org/10.24355/dbbs.084-202009161329-0

Algorithm 13 `getRecursivePC`($node$, $config$)

1: **if** $node$.`isAnd`() **then**
2: $children_+$:= `getChildrenWithVariableInConfig`($node$)
3: $children_-$:= `getChildrenWithoutVariableInConfig`($node$)
4: $\#overall := \prod_{child \in children_+ \cup children_-} \#overall_{child}$
5: $\#config := \prod_{child \in children_+} \#config_{child} \times \prod_{child \in children_-} \#overall_{child}$
6: **end if**
7: **if** $node$.`isOr`() **then**
8: $children$:= `getChildren`($node$)
9: $\#overall := \sum_{child \in children} \#overall_{child}$
10: $\#config := \sum_{child \in children} \#config_{child}$
11: **end if**
12: **if** $node$.`isPositiveLiteral`() **then**
13: $\#overall := 1$
14: **if** $f_{node} \in$ `getIncluded`($config$) **then**
15: $\#config := 1$
16: **end if**
17: **if** $f_{node} \in$ `getExcluded`($config$) **then**
18: $\#config := 0$
19: **end if**
20: **end if**
21: **if** $node$.`isNegativeLiteral`() **then**
22: $\#overall := 1$
23: **if** $f_{node} \in$ `getIncluded`($config$) **then**
24: $\#config := 0$
25: **end if**
26: **if** $f_{node} \in$ `getExcluded`($config$) **then**
27: $\#config := 1$
28: **end if**
29: **end if**
30: **return** $\#overall$,$\#config$

https://doi.org/10.24355/dbbs.084-202009161329-0

after the CBDD is created [OBMS16]. We describe this procedure in detail later. However, the authors argue that CBDDs do not scale for large systems regarding memory and runtime of the translation [OBMS16, OBMS17, OGB+19].

Another proposal by Oh et al. [OGB+19] uses the DPLL-based #SAT solver sharpSAT for uniform random sampling. The procedure is shown in Algorithm 14. Given a feature $f \in FEAT_{FM}$ the number of valid configurations that do not contain f ($\#FM_{\neg f}$) is computed in line **3**. If $r \leq \#FM_{\neg f}$, f is not part of the sampled configuration as seen in lines **4-5**. Otherwise, f is included and r is adjusted $r = r - \#FM_{\neg f}$ for the following iterations as seen in lines **6-8**. This procedure is repeated until every feature is either included or selected in the configuration. By construction, it is not possible to assign a variable such that the resulting number of valid configurations under that assignment is zero as zero is always smaller than the adapted random number. Thus, every resulting configuration is valid. Furthermore, different random numbers always result in different configurations and the chance that a specific configuration appears in the sample is not dependent on the order of variables. The described procedure ensures a one-to-one mapping between integers $i \in [1, \#FM]$ and the valid configurations as every random number $r \in [1, \#FM]$ results in a different valid configuration. Assuming unbiased random numbers, it follows that each configuration has appears in the sample with the same probability. The procedure requires up to n #SAT calls if n is the number of features. To decrease the runtime, Oh et al. [OGB+19] also consider further optimizations, which we discuss later.

Algorithm 14 `UniformRandomSampling`(FM, r) adapted from [OGB+19]

```
1:  config := ∅
2:  for f ∈ FEATS_FM do
3:      count := #SAT(CNF_FM ∧ (⋀_{v∈config} v) ∧ ¬f)
4:      if r ≤ count then
5:          config := config ∪ {¬f}
6:      else
7:          config := config ∪ {f}
8:          r := r − count
9:      end if
10: end for
11: return config
```

Optimizations

As the base algorithm repeatedly computes the number of remaining configurations of a partial configuration, each optimization that is applicable for partial configurations is applicable for uniform random sampling. This also included the optimizations for feature models and commonality that are applicable for partial configurations. In the following, we describe optimizations and algorithms that specifically optimize uniform random sampling.

Boolean Constraint Propagation

Boolean constraint propagation (BCP) can be performed to simplify the formula by propagating the assignment of a variable to all clauses [CK05a]. Typically, a tool

https://doi.org/10.24355/dbbs.084-202009161329-0

performs BCP if a unit clause is found. A unit clause only contains one literal. In order to satisfy the clause a positive literal has to be assigned $\top$ and a negative literal $\bot$. The assignment of the variable can then be propagated to the other clauses which potentially results in new unit clauses. After including or excluding a feature in Algorithm 14 (lines **5** and **7**), BCP can be performed to include or exclude further features. This potentially saves a high number of #SAT calls. The features corresponding to a variable that appears in a unit clause can also be directly included. This potentially saves a high number of #SAT calls.

After some iterations of the algorithm provided by Oh et al. [OGB+19], it is possible that boolean constraint propagation results in an empty formula (i.e., every clause is satisfied). In this case, every unassigned variable can be freely assigned resulting in 2^n remaining valid configurations if n is the number of unassigned variables. Thus, it is not necessary anymore to compute the number of remaining valid configurations. Oh et al. propose the following procedure to resolve the remaining assignments. For each feature f_i the current random number $r_{i+1} = r_i/2$ is halved. If $r_{i+1}\%2 = 0$, f is excluded from the configuration. Otherwise, it is included. It follows that each remaining feature has a 50% to be included and each different r results in a different configuration. This potentially reduces the number of required #SAT calls and, thus, the runtime [OGB+19].

Variable Ordering

The #SAT calls required for uniform random sampling can also benefit from alternative variable orderings for the DPLL traversal like proposed in Section 4.1. If a feature is included or included in the configuration and, thus, a variable is assigned, the formula can be simplified with boolean constraint propagation. The adapted formula is then used for the following iterations. Therefore, processing variables first that simplify the formula more effectively potentially reduces the runtime required for the #SAT in the following iterations and, thus, the overall runtime.

Oh et al. [OGB+19] propose using cube-and-conquer to accelerate the sampling process which we already considered for computing the number of valid configurations in Section 4.1. Cube-and-conquer computes variables that eliminate the highest number of clauses if boolean constraint propagation is applied. If these variables are processed first, the following #SAT calls are accelerated and the number of required #SAT calls is potentially reduced. Thus, this technique is especially beneficial for algorithms that re-use adapted formulas, like uniform random sampling.

Propagating Results from Other Analyses

We propose to reuse information from other analyses to reduce the number of required #SAT calls for uniform random sampling. This idea is similar to the optimization that propagates results from other analyses in Section 4.2. A dead/core feature is part of no/all valid configurations, respectively. Therefore, a dead/core feature can be excluded/included without an additional #SAT call. Furthermore, atomic sets can be used for further optimization. Let A be an atomic set and $f, g, h \in A$ features. If f is added to a configuration, g and h can be added without further #SAT calls.

https://doi.org/10.24355/dbbs.084-202009161329-0

d-DNNF Traversal

We propose to use d-DNNF query to evaluate the current partial configuration instead of a #SAT call to perform uniform random sampling. In Section 4.3, we proposed an algorithm that exploits the properties of an d-DNNF to compute the number of remaining valid configurations. This can be applied for uniform random sampling by simply replacing the #SAT calls with an according d-DNNF query in Algorithm 14. The d-DNNF just needs to be computed once at the start of the procedure. Additionally, we can exploit the extraction of core and dead features we introduced for traversing d-DNNF to compute commonalities in Section 4.2.

Uniform random sampling using #SAT typically requires a large number of #SAT calls. A single sample requires up-to $|FEATS_{FM}|$ #SAT calls in the worst case [OGB+19]. Typically, multiple samples are required for effective testing [AMS+18, VAHT+18]. Thus, it may be beneficial to compute the d-DNNF once for polynomial queries during the sampling process. Overall, we aim to reduce the number of requires queries by using the knowledge about core and dead features and reduce the required runtime of the queries by using a d-DNNF.

Sharma et al. [SGRM18] also used d-DNNFs to perform uniform random sampling. The idea is to find n distinct samples within a single d-DNNF traversal starting from the root by recursively deciding how many samples are extracted from the children of an Or-node. After a recursion step, n is adapted. In the following, we the adapted n for a specific node as n'. As the traversal always ends in a literal node, the procedure results in concrete valid configurations induced by the included literals. Algorithm 15 shows the procedure. For And-nodes, each child induces solutions that only contain a distinct subset of the variables. The procedure is called recursively for every child of the node resulting in n samples for each child as seen in lines **4-6**. These can be merged to acquire n' solutions with all variables contained in the parent And-node as seen in line **7**. As an example, consider an And-node with two children with $n = 2$ which return $x_1 = \{A, B\}, x_2 = \{\neg A, B\}$ and $y_1 = \{C, \neg D\}$, $y_2 = \{\neg C, \neg D\}$, respectively. These sets are merged resulting in $x_1 \cup y_1 = \{A, B, C, \neg D\}$ and $x_2 \cup y_2 = \{A, B, C, \neg D\}$ as samples for the parent node. For Or-nodes, the procedure picks how many solutions are derived from a child node which is shown in lines **12-16** in the following way. First, the probability with which a solution belongs to the child is computed via $\frac{\#child}{\#node}$ as seen in line **13**. $\#child$ corresponds to the number of solutions induced by the child and $\#node$ corresponds to the number of solutions induced by its parent. Second, the number of samples that should be derived from the child is computed as seen in line **14** using a binomial distribution over the probability and n'. Third, the procedure is recursively called for each child with the computed number of samples as seen in line **15**. As an example consider an Or-node with two children x, y and $n' = 10$. x and y induce 400 and 600 solutions respectively. Therefore, the procedure chooses four solutions from x and six solutions from y on average. For a Literal-node, n' samples are created and the literal corresponding to the node is added to each sample as seen in lines **20-25**.

We expect that the same optimizations can be applied to both d-DNNF-based algorithms for uniform random sampling as for commonality and partial configurations

https://doi.org/10.24355/dbbs.084-202009161329-0

as they should accelerate the compilation to d-DNNF. However, using knowledge about core and dead features is not applicable for the second procedure without adaptations.

Algorithm 15 `recursiveUrs`$(node, n)$ adapted from Sharma et al. [SGRM18]

```
1: if node.isAnd() then
2:     children := getChildren(node)
3:     partialSamples := []
4:     for child in children do
5:         partialSamples.add(recursiveUrs(child, n))
6:     end for
7:     return mergePartialSamples(partialSamples)
8: end if
9: if node.isOr() then
10:    samples = []
11:    children := getChildren(node)
12:    for child in children do
13:        probability_child = #child / #node
14:        n_child = getNumberOfSamples(n, probability_child)
15:        samples.add(recursiveUrs(child, n_child))
16:    end for
17:    return samples
18: end if
19: if node.isLeaf() then
20:    samples = []
21:    for i in [1, n] do
22:        sample = ∅
23:        sample.add(node)
24:        samples.add(sample)
25:    end for
26:    return samples
27: end if
```

Traversal of Counting BDDs

Oh et al. [OBMS16] propose using a counting BDD (CBDD) for uniform random sampling. The difference to a regular BDD is that an edge outgoing from f indicates the number of valid assignments that remain after the assignment of f. For example, the number on the low edge indicates the remaining satisfying assignment if $\alpha(f) = \bot$. After computing the CBDD, a valid configuration can be created by the single traversal of a single path in the CBDD. Thus, creating a configuration has a time complexity of $O(|FEATS_{FM}|)$.

Algorithm 16 describes the procedure. Given a random number $r \in [1, \#FM]$, the procedure starts at the top node. In this paragraph, we reference to the number of remaining satisfying assignment when taking the high/low edge of f as $\#\text{high}_f/\#\text{low}_f$. At each variable node, if $r > \#\text{low}_f$, the high edge is taken and $r = r - \#\text{low}_f$is

https://doi.org/10.24355/dbbs.084-202009161329-0

adapted as seen in lines **10-13**. Otherwise, the low edge is taken as seen in lines **14-16**. Often the assignment of a variable has no impact on the satifisfiability of a path and is omitted as seen in lines **4-9**. In this case, the variable is included if $r\%2 = 1$ as seen in lines **5-7**. Afterwards, r is halved even if the variable was excluded to ensure that r is always smaller than the number of remaining valid configurations as seen in line **8**. The procedure can only result in valid configurations. In addition, the configurations are uniformly distributed [OBMS16]. Analogously to d-DNNFs, we expect that we can apply the optimizations that reduce the required runtime for #SAT calls to the compilation from CNF to BDD.

Algorithm 16 `BDDURS`(*config*, r, f) adapted from [OBMS16]

```
1:  if reached1Terminal() then
2:      return
3:  end if
4:  if isOmittedInPath(f) then
5:      if r%2 = 1 then
6:          config.add(f)
7:      end if
8:      r := r/2
9:  end if
10: if r > #low_f then
11:     config.add(f)
12:     r := r−#low_f
13:     BDDURS(config, r, f.high)
14: else
15:     BDDURS(config, r, f.low)
16: end if
17: return config
```

Recap

We provided six optimizations specific for uniform random sampling. The first three, namely boolean constraint propagation, variable ordering, and propagating results from other analyses accelerate the described base algorithm by reducing the number of required #SAT calls. In addition, boolean constraint propagation also reduces the required runtime of the #SAT calls, potentially more effective with certain variable orderings (e.g., cube-and-conquer). The latter three optimizations employ knowledge compilation to reduce the time required for #SAT queries. Two of those optimizations use d-DNNFs and the other one counting BDDs.

4.5 Summary

In this chapter, we presented algorithms and optimizations for feature-model analyses that are dependent on #SAT technology. First, we discussed computing the number of valid configurations of a feature model. We **(1)** showed how to compute the number of valid configurations with #SAT solver, **(2)** discuss four ideas to

https://doi.org/10.24355/dbbs.084-202009161329-0

reduce the required runtime for a #SAT call, and **(3)** present an algorithm for incrementally computing results of a following version of a feature model by computing the number of added and removed valid configurations.

We also considered algorithms and optimizations for computing the commonality of features. We showed how to compute the commonality of a feature with: (1) a regular #SAT solver, **(2)** an adaptation of DPLL, **(3)** an algorithm that evaluates the feature-tree and cross-tree constraints separately, **(4)** a d-DNNF. Furthermore, we proposed to skip #SAT calls for some features by using results from other analyses. In addition, the optimizations considered for feature models are also applicable for commonality.

We presented two algorithms for computing the number of remaining valid configurations of a partial configuration. We **(1)** show how to compute the number of remaining valid configurations of a partial configuration with a #SAT solver and **(2)** provide an algorithm that computes the result with a d-DNNF traversal. In addition, all optimizations considered for feature models are applicable for partial configurations. The optimizations for commonality require further adaptations for the usage on partial configurations.

At last, we discuss uniform random sampling. We **(1)** show how to perform uniform random sampling using a #SAT solver, **(2)**, discuss two optimizations to reduce the number of required #SAT calls and the runtime required for the calls, and **(3)** propose two algorithms that exploit the properties of a d-DNNF to perform uniform random sampling. The majority of the algorithms is based on repetitive #SAT calls on a partial configurations. Thus, the same optimizations are applicable for uniform random sampling.

Each application described in Chapter 3 can be compute using on of the four analyses considered in this chapter. Thus, every described algorithm and optimization can be used to compute results for multiple applications for feature models. In addition, various optimizations can be used in parallel to further reduce the required runtimes. In Chapter 5, we describe the implementation of the following algorithms. For every analysis, we implement the base algorithm. For commonality, partial configurations, and uniform random sampling, we implement the described algorithms based on d-DNNF. In addition, we implement the propagation of results from other analyses for commonality and uniform random sampling. The algorithms described in this chapter build the base for our empirical evaluation as we evaluate the listed implementations.

https://doi.org/10.24355/dbbs.084-202009161329-0

5. Implementation

In this chapter, we describe the implementation of algorithms that are dependent on #SAT and used to compute the number of valid configurations of a feature model, the commonality of features, the number of remaining valid configurations of a partial configuration, and uniform random sampling. The goal is to enable the reader to understand the algorithms and design decisions in the implementations such that the reader could re-implement the algorithms. Additionally, we describe the integration of these algorithms in FeatureIDE [fea19]. This may be especially interesting for readers that aim to use the mentioned functionalities. Furthermore, we describe the benchmark framework used for our empirical evaluation, such that our experiments are comprehensible and the reader is able to perform new experiments.

We describe our implementations in Java [AGH05]. We implement several algorithms that each compute one of the following four analyses: 1) compute the number of valid configurations of a feature model, 2) compute the commonalities of features, 3) compute the number of remaining configurations of a partial configurations, 4) perform uniform random sampling. Furthermore, we present our d-DNNF engine that supports these four metrics and can be used with multiple d-DNNF compilers. To use the engine, the feature model needs to be translated once to d-DNNF. Afterwards, the four analyses can be run in polynomial time without computing the d-DNNF once again.

This chapter is separated in three topics. First, we provide implementations of the algorithms described in Chapter 4. Here, we describe the general idea to use #SAT solvers for the analysis. Afterwards, we present the concrete implementations for the different analyses and explain the implementation of the d-DNNF engine. Second, we describe integration in the feature modeling tool FeatureIDE. Third, we provide a description of the benchmark framework used for the empirical evaluation. To this end, we describe structure and important classes of the framework. Afterwards, we explain how to repeat the experiments of our empirical evaluation, perform other experiments, and add new solvers, algorithms, and feature models.

https://doi.org/10.24355/dbbs.084-202009161329-0

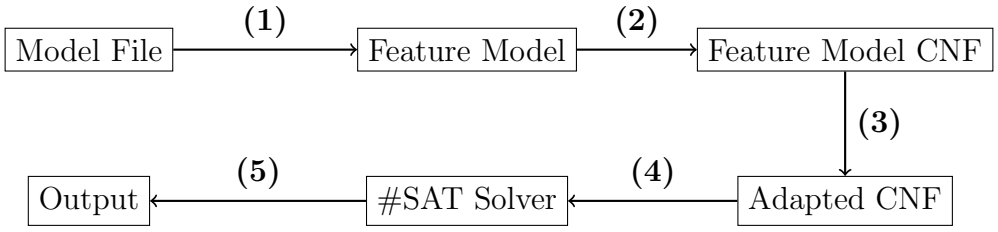

Figure 5.1: Overview General Procedure

5.1 Implementation of Applications

In this section, we describe the technical implementation of the algorithms presented in Chapter 4. We start by describing the procedure and tool chain that is shared by all the algorithms and we used to call #SAT on a feature model. To this end, we describe the translation from feature model to a propositional formula, the transformation to CNF, and the call of the #SAT solver. We do not consider the process of parsing an arbitrary configurable system to a feature model. On a high level, the analysis of a feature model with a #SAT solver is described in Figure 5.1. Also, Listing 5.1 shows the implementation of that procedure in Java.

```
1   // 1. Parse file
2     String modelName = FileUtils.getFileNameWithoutExtension(file);
3     IFeatureModel model = FMUtils.readFeatureModel(file);
4
5     // 2. Translate to CNF
6     FeatureModelFormula formula = new FeatureModelFormula(model);
7     CNF cnf = formula.getCNF();
8
9     // 3. Adaptations depending on specific algorithm
10    cnf = adaptCNFDependingOnAlgorithm(cnf);
11
12    // 4. Save CNF as DIMACS
13    DIMACSUtils.createTemporaryDimacs(cnf);
14
15    // 5. Execute Solver
16    binaryResult = solver.executeSolver(DIMACSUtils.
          TEMPORARY_DIMACS_PATH, timeout);
17    solverResult = solver.getResult(binaryResult.stdout);
18    results.put(modelName, solverResult.result.toString());
```

Listing 5.1: Implementation General Procedure

In **(1)**, the tool reads and parses a feature model in any format supported by FeatureIDE. FeatureIDE supports a variety of feature model formats. Most of them are based on xml. Thus, we provide the feature models in a .xml format that is supported by FeatureIDE and delegate the parsing to FeatureIDE. This part is shown in lines **1-3** in Listing 5.1.

In **(2)**, the tool translates the feature model to a CNF. FeatureIDE natively supports the translation of a feature model to a propositional formula and a conversion to CNF which is equisatisfiable and also preserves the same number of satisfying assignments [fea19]. The procedure is similar to the rules we introduced in Chap-

https://doi.org/10.24355/dbbs.084-202009161329-0

ter 2. Thus, we can just delegate the translation to FeatureIDE. This part is shown in lines **5-7** in Listing 5.1.

In **(3)**, the tool performs changes to the CNF according to the specific algorithm (e.g., add a unit clause of a feature to compute its commonality). The number of satisfying assignments of the resulting propositional formula is equal to the number of valid configurations of the feature model. Depending on the algorithm, the formula can now be adapted to analyze a specialization or generalization of the feature model. We discuss the required adapations to the formula for specific algorithms later. This part is shown in lines **9-10** in Listing 5.1.

In **(4)**, the tool stores the CNF in DIMACS format. Every #SAT solver we considered uses a CNF in the DIMACS format as input. FeatureIDE supports to save the CNF in DIMACS format. Thus, we also delegate this procedure to FeatureIDE. This part is shown in lines **12-13** in Listing 5.1.

In **(5)**, the tool calls the selected #SAT solver with the new DIMACS file as input. We pass the path of the previously stored DIMACS file to our selected #SAT solver. After the solver terminates, we parse the command line output to get the number of satisfying assignments. This part is shown in lines **15-18** in Listing 5.1.

5.1.1 Algorithm Implementations

In this section, we describe the implementation specifics for the four considered analyses used for the empirical evaluation, namely computing the number of valid configurations of a feature model, computing the commonalities of features, the number of remaining configurations for a partial configuration, and performing uniform random sampling. We discuss algorithms that are dependent on exploiting the properties of a d-DNNF separately in the next section.

Number of Valid Configurations

The CNF computed in step **2** of Figure 5.1 is equivalent to the underlying feature model. Thus, invoking a #SAT solver with the CNF as input computes the number of valid configurations of the feature model. Therefore, we can skip step **3** as no adaptation of the CNF is required. We implemented the resulting procedure to compute the number of valid configurations of a feature model.

Commonality

For the commonalities of features, we implemented two algorithms for the evaluation besides the d-DNNF implementation. First, we implemented a naive approach that performs a #SAT call for each feature f using $F_{FM} \wedge f$ as input. This corresponds to the base algorithm Algorithm 8 for computing commonalities described in Section 4.2. Listing 5.2 shows a snippet of the used Java code. The currently iterated feature is conjuncted to the CNF in lines **2-6**. The #SAT solver is invoked with the adapted CNF as input in lines **8-11**. Finally in line **13**, the result of the #SAT solver is stored as commonality for the respective feature.

https://doi.org/10.24355/dbbs.084-202009161329-0

```
1     for (IFeature feat : model.getFeatures())  {
2       // 1. Adapt formula
3       CNF temp = cnf.clone();
4       String featName = feat.getName();
5       int varIndex = temp.getVariables().getVariable(featName);
6       temp.addClause(new LiteralSet(varIndex));
7
8       // 2. Invoke solver
9       DIMACSUtils.createTemporaryDimacs(temp);
10      binaryResult = solver.executeSolver(runner,DIMACSUtils.
            TEMPORARY_DIMACS_PATH, timeout);
11      solverResult = solver.getResult(binaryResult.stdout);
12
13      commonalities.put(featName, solverResult.result.toString());
14    }
```

Listing 5.2: Commonality Implementation

We also implement an adaptation that performs analyses that compute core, dead, and false-optional features to reduce the number of required #SAT calls. We delegate these analyses to FeatureIDE. If a feature is either core, dead, false-optional, or mandatory, the #SAT call is redundant and can be skipped to save time. In Section 4.2, we explain the details of the procedure. The non-naive approach corresponds to the optimization using result propagation of other analyses described in Section 4.2, it's implementation in Java is shown in Listing 5.3. The core, dead, and mandatory/false-optional features are handled in lines **4-6**, **8-10**, and **12-14**, respectively. `overallModelCount` stores the number of valid configurations for the feature model. Features that do not meet any of these conditions are handled in lines **16-28**. This behavior is equivalent to the naive procedure shown in Listing 5.2.

```
1  for (IFeature feat : model.getFeatures())  {
2    String featName = feat.getName();
3
4    // 1. Handle core features
5    if (coreFeatures.contains(feat))  {
6      commonalities.put(featName, overallModelCount);
7
8    // 2. Handle dead features
9    }  else if (deadFeatures.contains(feat))  {
10     commonalities.put(featName, BigInteger.ZERO);
11
12   // 3. Handle mandatory and false-optional features
13   } else if (feat.getStructure().isMandatory() ||
         falseOptionalFeatures.contains(feat))  {
14     commonalities.put(featName, commonalities.get(FMUtils.getParent(
           feat).getName()));
15
16   // 4. Handle other features (analogous to Listing 5.2)
17   } else  {
18     // 4.1 Adapt formula
19     CNF temp = cnf.clone();
20     int varIndex = temp.getVariables().getVariable(featName);
21     temp.addClause(new LiteralSet(varIndex));
22
23     // 4.2 Invoke solver
24     DIMACSUtils.createTemporaryDimacs(temp);
```

https://doi.org/10.24355/dbbs.084-202009161329-0

```
25      binaryResult = solver.executeSolver(runner,DIMACSUtils.
            TEMPORARY_DIMACS_PATH, timeout);
26      solverResult = solver.getResult(binaryResult.stdout);
27      commonalities.put(featName, solverResult.result.toString());
28    }
29  }
```

Listing 5.3: Commonality Adaptation

Partial Configurations

For partial configurations, we implement Algorithm 12 described in Section 4.3 and the computation using d-DNNFs which we describe separately later. The procedure is similar to the base implementation of computing commonalities. Given a partial configuration $C = (FM,I,E)$ with the set of included features I and set of excluded features E, #SAT is called with $F_{FM} \wedge \bigwedge_{i \in I} i \wedge \bigwedge_{e \in E} \neg e$ as input. Listing 5.4 shows a snippet of the used Java code. The included and excluded features are conjuncted in lines **1-6** and **8-13**, respectively. #SAT is invoked with the adapted CNF as input in lines **15-18**.

```
1  // 1. Conjunct included features
2  for (IFeature feat : config.getSelectedFeatures())  {
3    String featName = feat.getName();
4    int varIndex = cnf.getVariables().getVariable(featName);
5    temp.addClause(new LiteralSet(varIndex));
6  }
7
8  // 2. Conjunct negation of excluded features
9  for (IFeature feat : config.getUnSelectedFeatures())  {
10   String featName = feat.getName();
11   int varIndex = cnf.getVariables().getVariable(featName);
12   temp.addClause(new LiteralSet(-varIndex));
13 }
14
15 // 3. Perform #SAT call
16 DIMACSUtils.createTemporaryDimacs(cnf);
17 binaryResult = solver.executeSolver(runner,DIMACSUtils.
       TEMPORARY_DIMACS_PATH, timeout);
18 solverResult = solver.getResult(binaryResult.stdout);
```

Listing 5.4: Partial Configurations Implementation

Uniform Random Sampling

For uniform random sampling, we implement the naive Algorithm 14 and an adaptation in which we propagate use results of previous analyses (e.g., core and dead features) to reduce the number of required #SAT calls. Listing 5.5 shows a snippet of the used Java code. Beginning on line **2**, the procedure computes the number of valid configurations for the feature model. On line **8**, a random number between zero and the `numberOfSolutions` is generated. This number is used to generate the configuration by iterating over the features and including or excluding the feature from the configuration C which is implemented in lines **16-40**. For each feature, the variable corresponding to the feature is negated and conjuncted to a temporary

https://doi.org/10.24355/dbbs.084-202009161329-0

copy of the CNF and the temporary CNF is stored as DIMACS in lines **19-23**. Then, the selected #SAT solver computes the number of valid configurations with respect to the current configuration in lines **26-27**. If the feature should be part of the configuration according to the rules described in Chapter 4, it is added to the configuration, added to the CNF for the next iterations, and the random number is adapted in lines **34-39**. Otherwise, the negated literal is added to the CNF for the next iterations in line **31**.

```
1  // 1. Compute number of valid configurations
2  IFeatureModel model = FMUtils.readFeatureModel(file);
3  FMUtils.saveFeatureModelAsDIMACS(model, DIMACSUtils.
       TEMPORARY_DIMACS_PATH);
4  binaryResult = solver.executeSolver(runner,DIMACSUtils.
       TEMPORARY_DIMACS_PATH, timeout);
5  BigInteger numberOfSolutions = solver.getResult(binaryResult.stdout).
       result;
6
7  // 2. Get random number
8  BigInteger randomNumber = getRandomNumber(0, numberOfSolutions);
9
10 // 3. Get CNF
11 FeatureModelFormula formula = new FeatureModelFormula(model);
12 CNF cnf = formula.getCNF();
13 List<String> includedFeatures = new ArrayList<>();
14
15 // 4. Perform uniform random sampling
16 for (IFeature feat : model.getFeatures())  {
17
18   // 4.1 Adapt formula
19   CNF temp = cnf.clone();
20   String featName = feat.getName();
21   int varIndex = temp.getVariables().getVariable(featName);
22   temp.addClause(new LiteralSet(-varIndex));
23   DIMACSUtils.createTemporaryDimacs(temp);
24
25   // 4.2 Execute #SAT
26   binaryResult = solver.executeSolver(runner,DIMACSUtils.
         TEMPORARY_DIMACS_PATH, timeout);
27   solverResult = solver.getResult(binaryResult.stdout);
28
29   // 4.3a Include feature
30   if (solverResult.result.compareTo(randomNumber) >= 0)  {
31     cnf.addClause(new LiteralSet(-varIndex));
32
33   // 4.3b Exclude feature
34   } else {
35       includedFeatures.add(featName);
36       cnf.addClause(new LiteralSet(varIndex));
37       randomNumber = randomNumber.subtract(solverResult.result);
38     }
39   }
40 }
41 return includedFeatures;
```

Listing 5.5: Uniform Random Sampling Naive

https://doi.org/10.24355/dbbs.084-202009161329-0

In the adaptation shown in Listing 5.6, the CNF is pre-processed using knowledge from the feature model. Prior starting to create samples, the set of core, dead, and false-optional are computed. If the feature is core, it is immediately added to the configuration and added to the CNF as a unit clause in lines **8-10**. Even though a feature is added, the random number does not need to be adapted as $\#(F_{FM} \wedge \neg f) = 0$ always holds for a core feature f. This behavior is similar for mandatory and false-optional features whose parents are part of the configuration, as $\#(F_{FM} \wedge p \wedge \neg c) = 0$ always holds for a mandatory/false-optional child c of p. The procedure for false-optional and mandatory features is shown in lines **19-21** A dead feature is never part of a valid configuration. Thus, it is not added to the configuration and a negative literal corresponding to the feature is conjuncted to the CNF in lines **12-14**. The same procedure is performed for any feature whose parent has not been added in lines **16-17**, as it cannot be part of this configuration either. Here, it is important that a parent is always processed before its children, as a non-core child which is processed before its parent could never be part of the sample. For all feature that do not meet any of the above described conditions, the naive procedure presented in Listing 5.5 is used.

```
1  FeatureModelFormula formula = new FeatureModelFormula(model);
2  CNF cnf = formula.getCNF();
3  List<String> includedFeatures = new ArrayList<>();
4  for (IFeature feat : model.getFeatures())  {
5    String featName = feat.getName();
6    int varIndex = cnf.getVariables().getVariable(featName);
7
8    // 1. Handle core features
9    if (coreFeatures.contains(featName))  {
10     includedFeatures.add(featName);
11     cnf.addClause(new LiteralSet(varIndex));
12
13   // 2. Handle dead features
14   } else  if (deadFeatures.contains(featName))  {
15     cnf.addClause(new LiteralSet(-varIndex));
16   }
17
18   // 3. Handle features whose parents are not included
19   } else if (!includedFeatures.contains(feat.getParent().getName())) {
20     cnf.addClause(new LiteralSet(-varIndex));
21
22   // 4. Handle mandatory and false-optional features
23   } else if (includedFeatures.contains(feat.getParent().getName()) && (
       feat.isMandatory() || falseOptionals.contains(featName))) {
24     includedFeatures.add(featName);
25       cnf.addClause(new LiteralSet(varIndex));
26
27   // 5. Handle other features
28   } else {
29     CNF temp = cnf.clone();
30     temp.addClause(new LiteralSet(-varIndex));
31     DIMACSUtils.createTemporaryDimacs(temp);
32     binaryResult = solver.executeSolver(runner,DIMACSUtils.
         TEMPORARY_DIMACS_PATH, timeout);
33     solverResult = solver.getResult(binaryResult.stdout);
34     if (solverResult.result.compareTo(randomNumber) >= 0)  {
```

https://doi.org/10.24355/dbbs.084-202009161329-0

```
35          cnf.addClause(new LiteralSet(-varIndex));
36        } else  {
37          includedFeatures.add(featName);
38          cnf.addClause(new LiteralSet(varIndex));
39          randomNumber = randomNumber.subtract(solverResult.result);
40        }
41      }
42    }
43    return includedFeatures;
```

Listing 5.6: Uniform Random Sampling Propagation

All the code snippets presented above utilize the #SAT solvers including d-DNNF compilers as black boxes. In the following section, we present algorithms that instead re-use the output of d-DNNF compilers.

5.1.2 Exploitation of d-DNNFs

In this section, we describe our implementation for the analysis of feature models using deterministic decomposable negation normal forms (d-DNNF). We explained d-DNNFs in Chapter 2 and provided several algorithms for the analysis of feature models that exploit them in Chapter 4. On a high level, our procedure to exploit d-DNNFs works as follows. First, we create a CNF from a given feature model. Second, this CNF is used as input for an off-the-shelf compiler that translates the CNF to d-DNNF. Third, we parse the output of the compiler and build a d-DNNF in our own data structure. The parsed d-DNNF can then be used for several different analyses (e.g., uniform random sampling).

Compiler Format

Darwiche et al. [Dar04] introduced a format for the compiler `c2d` to store d-DNNFs in a format that is similar to DIMACS. Each sub-node only appears once in the file and multiple parents may reference it. This format is also used by the compilers `d4` [LM17] and `dSharp` [MMBH10]. Thus, parsing the format allows re-using the output of all the mentioned compilers. In this section, we explain the syntax of this format as specified by the `c2d` project [1].

The first line of the output file contains metadata of the d-DNNF, namely the number of nodes, edges, and variables. Each edge corresponds to a reference from a parent-node to its child. **nnf n e v** indicates that the d-DNNF contains n nodes, e edges, and v variables. Every other line represents a node that is a logical element, namely `And`, `Or`, `Literal`, `True`, or `False`.

A i x y z represents an `And`-node that has **i** (in this case $i = 3$) variables with the indices **x**, **y**, and **z**. **A 0** is a special case and represents a `True`-node.

O i j x y z represents an Or-node that has **j** (in this case **j** $= 3$) child nodes with the indices **x**, **y**, and **z**. For the value of **i**, there two cases: If **i** $= 0$, it can be ignored. Otherwise, the Or-node may only contain two children and **i** is a variable index corresponding to the variable v_i. **O i 2 x y** with **i** > 0 indicates that **x**

[1] http://reasoning.cs.ucla.edu/c2d/

https://doi.org/10.24355/dbbs.084-202009161329-0

only induces satisfying assignments that contain v_i (i.e., $\mathbf{x} \Rightarrow v_i$ holds) and $\mathbf{y}$ only induces satisfying assignments with $\neg v_i$ (i.e., $\mathbf{y} \Rightarrow \neg v_i$ holds). The node can be interpreted as the following formula: $x \vee y \wedge (v_i \Rightarrow \mathbf{x}) \wedge (v_i \Rightarrow \neg \mathbf{y})$. **O 0 0** represents a `False`-node.

L x represents a literal that corresponds to the variable **x**. The literal can either be positive **L x** or negative **L -x**. Listing 5.7 shows an example of a smooth d-DNNF in the specified format. It is equivalent to the CNF$(1 \vee \neg 2 \vee 3) \wedge (2 \vee 4 \vee 5) \wedge (4 \vee 6) \wedge 4$.

```
1  nnf 23 26 6
2  L 1
3  L 2
4  L 3
5  L 4
6  L 5
7  L 6
8  L -1
9  L -2
10 L -3
11 L -5
12 L -6
13 O 2 2 1 7
14 O 3 2 2 8
15 O 5 2 4 9
16 O 6 2 5 10
17 A 2 1 2
18 A 2 7 12
19 O 2 2 16 15
20 A 2 6 17
21 A 3 0 12 11
22 O 1 2 19 18
23 A 2 3 20
24 A 3 21 14 13
```

Listing 5.7: Example d-DNNF in c2d format

Analysis Data Structure

In this section, we describe the data structure we develop in which we store the d-DNNF given by the compiler. We use this structure for the empirical evaluation of the d-DNNF based algorithms and integrated it into FeatureIDE. Additionally, we describe the translation from the solver output to our data structure.

The class `Ddnnf`, shown in Figure 5.2 represents the entire d-DNNF using a list of nodes that each references its child nodes by index. These nodes are represented by the abstract class `dDnnfNode`. A node can be an instance of either `And`, `Or`, `PositiveLiteral`, `NegativeLiteral`, `True`, or `False`. For each node N, the number of satisfying assignments for the subtree that has N as root is stored in `overallModelCount`. This count is reused by every algorithm. All algorithms temporarily update the model count of the nodes depending on input and algorithms this updated model count is stored in `currentModelCount`. Often the value of a node does not change during an algorithm, this is indicated by the algortihms via setting `changedValue` to **false**. This information can be used by the parents of the node to skip redundant computations and save time. The three operations

https://doi.org/10.24355/dbbs.084-202009161329-0

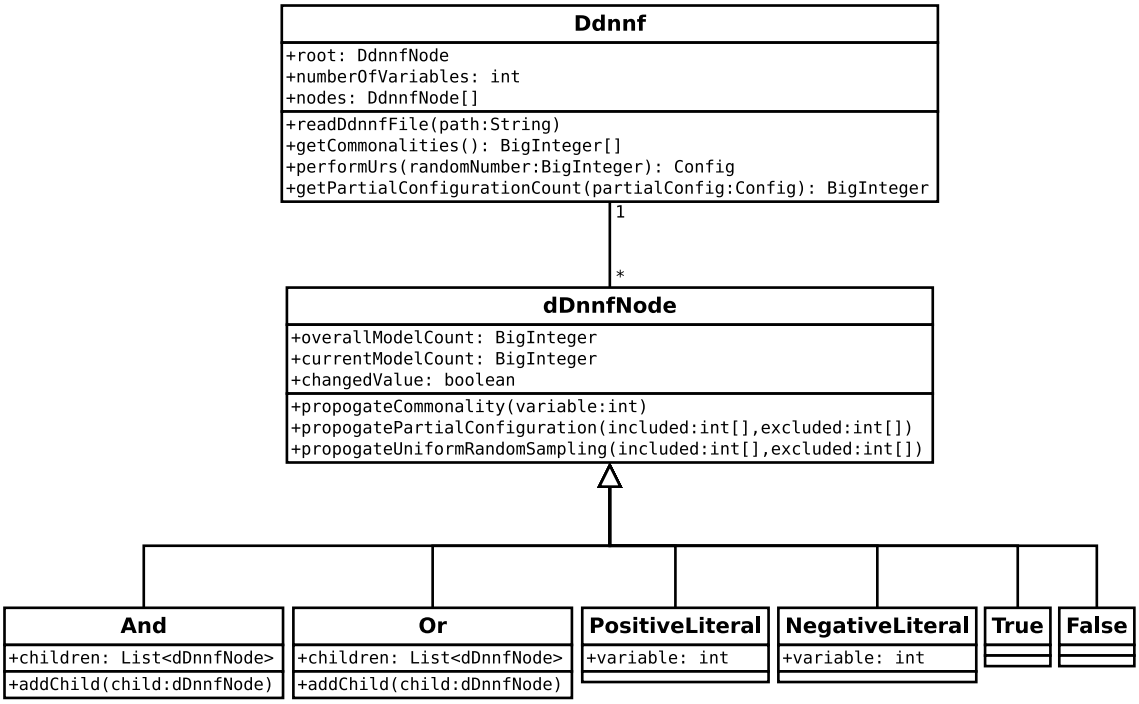

Figure 5.2: Class Diagram for d-DNNF Structure

`propogateCommonality()`, `propogatePartialConfiugration()`, and `propogate-UniformRandomSampling()` are used for the specific algorithms and are described later.

`And` additionally stores its children as a list of `dDnnfNode`. The node is initialized with a `overallModelCount` of one. When a child is added, the `overallModelCount` is multiplied with the `overallModelCount` of the child. Thus, after all children are added, the `overallModelCount` is the product of the children's `overallModelCount`.

`Or` stores its children as a list of `dDnnfNode`, like `And`. The node is initialized with an `overallModelCount` of zero. When a child is added, the `overallModelCount` of the child is added to `overallModelCount`. Thus, after all children are added, the `overallModelCount` is the sum of the children's `overallModelCount`.

`PositiveLiteral` and `NegativeLiteral` both correspond to one variable in the formula. Both nodes are initialized with an `overallModelCount` of one.

`True` is initialized with an `overallModelCount` and a `currentModelCcount` of one. These values always stay the same. `False` behaves the same way, but both values are initialized with zero.

The used off-the-shelf compiler stores the d-DNNF in a text file in the format described in the previous paragraph. Excluding the initial line that contains meta data, a `dDnnfNode` is created for each line of the file according to the rule set shown in Table 5.1. i,j,k represent arbitrary numbers. x,y,z represent the indices of arbitrary variables.

In the end of the parsing procedure a list of all nodes is stored in `Ddnnf`. The indices in the list are equivalent to the line indices in the parsed file. For every node the `overallModelCount` is stored for later use in the algorithms.

https://doi.org/10.24355/dbbs.084-202009161329-0

Entry	Procedure
nnf i j k	Save k as number of variables
A i x y z	Create `And` and add nodes with index x|y|z as children. `overallModelCount` $= \prod_{c \in \{x,y,z\}}$ `overallModelCount`$_c$
O i j x y z	Create `Or` and add nodes with index x|y|z as children. `overallModelCount` $= \sum_{c \in \{x,y,z\}}$ `overallModelCount`$_c$
L x	Create `PositiveLiteral` with `variable` = x `overallModelCount` = 1
L -x	Create `NegativeLiteral` with `variable` = x `overallModelCount` = 1
A 0	Create `True` `overallModelCount` = 1
O 0 0	Create `False` `overallModelCount` = 0

Table 5.1: d-DNNF Parsing Rules

Algorithms

In this section, we describe the implementations for computing the commonality of features, the number of remaining configurations for a partial configuration, and uniform random sampling by exploiting a d-DNNF. The algorithms were already described in Chapter 4.

For every algorithm, the `currentModelCount` of nodes is updated iteratively, given an assumption (e.g., to compute the commonality of feature f, the assumption $\alpha(f) = \top$ would be used). `overallModelCount` is constant and corresponds to the number of satisfying assignments of the original formula without any additional assumptions. During parsing, we preserve the order of nodes specified in the d-DNNF file given by the compiler. This ensures that the index of a child is always smaller than the index of its parent. Thus, if the procedure reaches a node, its children's `currentModelCount` are already updated according to the algorithm. Typically, the `currentModelCount` differs to the `overallModelCount` for only a subset of all nodes. It is not necessary to re-compute the `currentModelCount` if none of the children has changes. Each node holds a `changedValue`-flag that indicates whether its value changed during the current traversal of the d-DNNF. After the traversal, the `currentModelCount` of the `root` holds the result of the algorithm. The implementations for the algorithms mentioned above proceed similar and only differ in the procedure for `PositiveLiteral` and `NegativeLiteral`. In the following, we describe the behavior of the `DdnnfNode` subclasses `And` (Listing 5.8), `Or` (Listing 5.9), `True` (Listing 5.10), `False` (Listing 5.11), when propagating intermediate results for the different algorithms.

Listing 5.8 shows the procedure of propagating the model count for a new query at `And` nodes. First, the procedure checks whether any child changed its value by inspecting the boolean `changedValue` in lines **5-10**. Note, that a parent is always processed later than its children. If none of the children changed its value, the model count is not recomputed and `changedValue` is set to false. This potentially reduces the required time by a large margin as the runtime for a multiplication is

https://doi.org/10.24355/dbbs.084-202009161329-0

$O(n^2)$ where n is the number of digits and numbers sometimes reach more than 1000 digits [STS20]. Otherwise, the product of the children's counts is recomputed and `changedValue` is set to true in lines **14-28**. If the value changed for a child, its `currentModelCount` is used for the computation in lines **19-21**. If the value did not change, the child's `overallModelCount` is used in lines **24-25**.

```
1  public class And {
2
3    public void propagate() {
4
5      // 1. Check if any child changed
6      changedValue = false;
7      for (IterativeBUNode child : children) {
8        if (child.changedValue) {
9          changedValue = true;
10         break;
11       }
12     }
13
14     // 2. Recompute if any child changed
15     if (changedValue) {
16       currentModelCount = BigInteger.ONE;
17       for (IterativeBUNode child : children) {
18
19         // 2.a handle child that changed value
20         if (child.changedValue) {
21           currentModelCount =
22           currentModelCount.multiply(child.currentModelCount);
23
24         // 2.b handle child that did not change value
25         } else {
26           currentModelCount = currentModelCount.multiply(child.
                overallModelCount);
27         }
28       }
29     }
30 }
```

Listing 5.8: Procedure for And Nodes

Listing 5.9 shows the procedure of propagating the model count for a new query for an `Or` node. Equivalent to `And` nodes, the procedure first checks whether none of the children changed their values in lines **5-10**. If any of them changed their value, the sum of the children's counts is recomputed in lines **14-29** and `changedValue` is set to true. Otherwise, the value does not need to be recomputed and `changedValue` is set to false.

```
1  public class Or {
2
3    public void propagate() {
4
5      // 1. Check if any child changed
6      changedValue = false;
7      for (IterativeBUNode child : children) {
8        if (child.changedValue) {
9          changedValue = true;
```

https://doi.org/10.24355/dbbs.084-202009161329-0

```
10          break;
11        }
12      }
13
14      // 2. Recompute if any child changed
15      if (changedValue) {
16        currentModelCount = BigInteger.ONE;
17
18        for (IterativeBUNode child : children) {
19
20          // 2.a handle child that changed value
21          if (child.changedValue) {
22            currentModelCount =
23            currentModelCount.add(child.currentModelCount);
24
25          // 2.b handle child that did not change value
26          } else {
27            currentModelCount = currentModelCount.add(child.
                overallModelCount);
28          }
29        }
30      }
31    }
32  }
```

Listing 5.9: Procedure for Or Nodes

Listing 5.10 shows the procedure for a `True`-node for which the query is not relevant as the value for a `True`-node is always one. The behavior is similar for `False`-nodes. The procedure is shown in Listing 5.11. Here, the value is always zero independent of the query.

```
1  public class True {
2
3    public void propagate() {
4      currentModelCount = overallModelCount;
5      changedValue = false;
6    }
7  }
```

Listing 5.10: Procedure for True Nodes

```
1  public class False {
2
3    public void propagate() {
4      currentModelCount = overallModelCount;
5      changedValue = false;
6    }
7  }
```

Listing 5.11: Procedure for False Nodes

In contrast to the other nodes, `PositiveLiteral` and `NegativeLiteral` behave differently depending on the input and algorithm. The idea for each algorithm is to change the value `currentModelCount` of literals corresponding to the variable f, for an assumption $\alpha(f) = \top/\bot$. Let $\alpha(f) = \top$ be an assumption and `PositiveLiteral` and `NegativeLiteral` correspond to f. Then, the `currentModelCount` of `NegativeLiteral` is set to zero and `changedValue` is set to true. The

https://doi.org/10.24355/dbbs.084-202009161329-0

value for `PositiveLiteral` does not change as the node still induces one solution and `changedValue` is set to false. These changes are propagated using the presented procedures til the root is reached. The `currentModelCount` at the root node then is the result for the given assumption.

The procedures for `PositiveLiteral` are described in Listing 5.12. Uniform random sampling uses the propagation of a partial configuration in each iteration. Thus, we only consider methods for propagating values for commonality and partial configurations. When computing commonality the only possible adaption is conditioning a single variable f to be positive (i.e., $\alpha(f) = \top$). If the `PositiveLiteral` corresponds to this variable, the model count is still one. If it corresponds to another variable, nothing changes. Thus, the value for a `PositiveLiteral` never changes when computing commonality which is shown in lines **14-16**. For included variables of a partial configuration, the behavior is the same and implemented in lines **9-11**. However, the `PositiveLiteral` corresponds to an excluded variable, the model count is set to zero and `changedValue` is set to to **true** which can be see in lines **5-7**.

```
1  public class PositiveLiteral {
2
3    public void propogatePartialConfiguration(Set<Integer> included, Set<
         Integer> excluded) {
4
5      if (excluded.contains(variable)) {
6        currentModelCount = BigInteger.ZERO;
7        changedValue = true;
8
9      } else {
10       changedValue = false;
11     }
12   }
13
14   public void propagateCommonality(int variable) {
15     changedValue = false;
16   }
17
18
19 }
```

Listing 5.12: Procedures for PositiveLiteral Nodes

`NegativeLiteral` nodes behave similarly to `PositiveLiteral` nodes. For the commonality of a feature , if the considered feature corresponds to the same variable as `NegativeLiteral`, the value is changed to zero, as $\neg\top \equiv \bot$ which is shown in lines **16-18**. For every other variable, the model count stays the same at one in lines **20-22**. For partial configurations, the excluded features have no impact as seen in lines **9-11**. However, if the variable the `NegativeLiteral` corresponds to is part of the included features, the model count is changed to zero and changes value is set to **true** which is shown in lines **5-7**.

```
1  public class NegativeLiteral {
2
3    public void propogatePartialConfiguration(Set<Integer> included, Set<
         Integer> excluded) {
```

https://doi.org/10.24355/dbbs.084-202009161329-0

```
4
5       if (included.contains(variable)) {
6         currentModelCount = BigInteger.ZERO;
7         changedValue = true;
8
9       } else {
10        changedValue = false;
11      }
12    }
13
14    public void propagateCommonality(int variable) {
15
16      if (this.variable == variable) {
17        changedValue = true;
18        currentModelCount = BigInteger.ZERO;
19
20      } else {
21        changedValue = false;
22      }
23    }
24
25
26  }
```

Listing 5.13: Procedures for NegativeLiteral nodes

These procedures described above can now be used to compute commonalities of features, to compute the number of partial configurations and to perform uniform random sampling with a d-DNNF. Listing 5.14 shows the procedure for commonalities. If a feature has been found core or dead in lines **7** or **10**, a traversal through the d-DNNF is not required. Otherwise, `propogateCommonality(f)` is called for every node of the d-DNNF in lines **14-17**. Afterwards, the `root` holds the result in its `currentModelCount`.

```
1
2   public List<BigInteger> getCommonalities() {
3
4     List<BigInteger> commonalities = new ArrayList<>();
5     for (int i = 1; i <= numberOfVariables; i++) {
6
7       if (cores.contains(i)) {
8         commonalities.add(root.overallModelCount);
9
10      } else if (deads.contains(i)) {
11        commonalities.add(BigInteger.ZERO);
12
13      } else {
14        for (IterativeBUNode node : nodes) {
15          node.propagateCommonality(i);
16        }
17        commonalities.add(root.currentModelCount);
18      }
19    }
20    return commonalities;
21  }
```

Listing 5.14: Compute Commonalities with d-DNNF

https://doi.org/10.24355/dbbs.084-202009161329-0

Listing 5.15 shows the procedure for computing the remaining valid configurations for a partial configuration. The method inputs two sets of integer which correspond to the included and excluded variables. Here, the `propogatePartialConfigurationCount(included, excluded)` is called for every node in lines **4-6**. Afterwards, the `currentModelCount` holds the result.

```
1
2   public BigInteger getPartialConfigurationCount(Set<Integer> included,
        Set<Integer> excluded) {
3
4     for (IterativeBUNode node : nodes) {
5       node.propogatePartialConfigurationCount(included, excluded);
6     }
7     return root.currentModelCount;
8   }
```

Listing 5.15: Compute Partial Configurations with d-DNNF

Listing 5.16 shows the procedure for performing uniform random sampling with d-DNNF. The method inputs a `BigInteger` which is then mapped to a configuration. First, the required members `included`, `excluded`, and `randomNumber` are initialized in `ursInit(BigInteger)` in lines **11-16**. Second, the current variable is handled in `ursHandleNextVariable(int)` for each node in lines **18-31**. If a feature is core or dead, the d-DNNF does not need to be traversed. This is shown in lines **16-19**. Otherwise, the current variable is added to the excluded variables in line **25**. Then, the number of valid configurations of the partial configuration described by `included` and `excluded` is computed by invoking `getPartialConfigurationCount(included, excluded)` in line **26** which was described in Listing 5.15. If the returned number of satisfying assignments is smaller than the `randomNumber`, the `randomNumber` is updated, the variable is removed from `excluded` and added to `included` in lines **27-30**.

```
1
2  public Set<Integer> performUrs(BigInteger randomNumber) {
3
4    ursInit(randomNumber);
5    for (int i = 1; i <= numberOfVariables; i++) {
6      ursHandleNextVariable(i);
7    }
8    return included;
9  }
10
11 public void ursInit(BigInteger randomNumber) {
12
13   included = new HashSet<>();
14   excluded= new HashSet<>();
15   this.randomNumber = randomNumber;
16 }
17
18 public void ursHandleNextVariable(int variableIndex) {
19
20   if (cores.contains(variableIndex)) {
21     currentConfig.add(variableIndex)
22   } else if (deads.contains(variableIndex)) {
23
```

https://doi.org/10.24355/dbbs.084-202009161329-0

```
24    } else {
25      excluded.add(variableIndex);
26      BigInteger result = getPartialConfigurationCount(included, excluded
            );
27      if (result.compareTo(randomNumber) < 0) {
28        randomNumber = randomNumber.subtract(result);
29        excluded.remove(variableIndex);
30        included.add(variableIndex);
31      }
32    }
33  }
```

Listing 5.16: Uniform Random Sampling with d-DNNF

5.2 Integration into FeatureIDE

In this section, we describe the integration of the devised algorithms in FeatureIDE. After this section, the reader should understand the usage in FeatureIDE and comprehend the implementation which is available in our FeatureIDE fork[2]. The implementation is based on the previously described d-DNNF engine. First, we give a short introduction into FeatureIDE. Second, we describe the integration of the d-DNNF engine. Third, we present the integration of the four algorithms.

FeatureIDE

FeatureIDE is an integrated development environment used for developing feature-oriented software [TKB+14]. It is based on the IDE eclipse but also offers a headless library for feature modeling. The source code for the eclipse plugin and the library is available at the GitHub repository of the project[3].

FeatureIDE supports counting the number of valid configurations of a feature model and a partial configuration. However, both implementations use SAT solvers and are based on blocking clauses which work as follows. After finding a satisfying full assignment with the SAT solver, a term, with as many literals as there are features, is created that represents the assignment. Then, this term is negated and conjuncted to the CNF representing feature model as the blocking clause. For example, suppose solver returns a satisfying assignment $\{A, B, \neg C, \neg D\}$. The term representing the assignment is $T = A \wedge B \wedge \neg C \wedge \neg D$ and the result blocking clause is $\neg T = \neg A \vee \neg B \vee C \vee D$. The formula resulting from conjuncting the blocking clause to the current formula is then used as input for the next SAT call which returns another solution as long as there is one. The described procedure can be repeated until no satisfying assignment is left to count the number of valid configurations. However, this requires $\#FM$ SAT calls. Additionally, the CNF is continually growing during the procedure which may even cause later runs to be slower. Therefore, the described algorithm only scales to small feature models.

For sampling, FeatureIDE supports a variety of algorithms. Most of them are based on t-wise interaction coverage (e.g., Chvatal [Chv79] and Incling [AHKT+16]). One algorithm also creates random samples, but without a guarantee for a uniform distribution. Currently, FeatureIDE does not support computing commonality of features.

[2] https://github.com/SundermannC/FeatureIDE/tree/ma_ddnnf
[3] https://github.com/FeatureIDE/FeatureIDE

https://doi.org/10.24355/dbbs.084-202009161329-0

Integration d-DNNF

We implement the d-DNNF structure described in Section 5.1.2 in FeatureIDE. We save the source code in the package `de.ovgu.featureide.fm.core.analysis.ddnnf`. For the compilation of a CNF to d-DNNF, we used the `dSharp` compiler which we added as a binary in **/plugins/de.ovgu.featureide.fm.core/lib**. In the following, we describe how we integrate the four computations in FeatureIDE. This is supposed to enhance the reader to use the new computations in FeatureIDE and comprehend the underlying implementation.

Integration Number of Valid Configurations

FeatureIDE offers a view that provides syntactical (e.g., number of features and constraints) and semantical (e.g., number of core, dead, and false-optional features) statistics about the currently loaded feature model. Figure 5.3 shows a screenshot of the statistics view in FeatureIDE. The semantical statics also contain an entry for counting the number of valid configurations that currently uses procedure described above. As computing the number of valid configurations is an expensive operation, the computation is only available on demand by double-clicking the corresponding entry.

The existing interface provided a CNF representing the feature model and a timeout that can be set by the user. We replaced the underlying analysis in the class `CountSolutionsAnalysis` with a #SAT based computation. Thus, we used `dSharp` as a model counter to compute the number of valid configurations.

Integration Commonality

FeatureIDE provides a graphical editor for feature models which can be used to create and edit a feature model. For example, features or constraints can be added and deleted. The editor also allows to compute the following anomalies: void model, core, dead, false-optional features, tautological, and redundant constraints. Furthermore, it is possible to compute explanations for the anomalies. These analyses can be manually run or automatically after a change to the feature model. This is enabled by a drop-down shown in Figure 5.4.

The computation of commonalities of features is currently not integrated in FeatureIDE. We decided to integrate commonalities in the editor for feature models. We added the entry *Compute Commonalities* to the drop-down menu used for analyses which is shown in Figure 5.5. After running the analysis, the commonality of features is visualized in two ways. First, each feature is colored with gradations that indicate its commonality. For example, a core feature (i.e., commonality of one) is colored in dark green and a feature with a commonality $0.25 < c \leq 0.5$ is colored in yellow as seen in Figure 5.6 for features `Carbody` and `CD`, respectively. Additionally, the actual value is displayed in the tooltip of the feature as seen in Figure 5.7.

Integration Partial Configurations

FeatureIDE provides a graphical editor to derive configurations. The user can select and deselect features while the tool automatically performs selection propagation. In

https://doi.org/10.24355/dbbs.084-202009161329-0

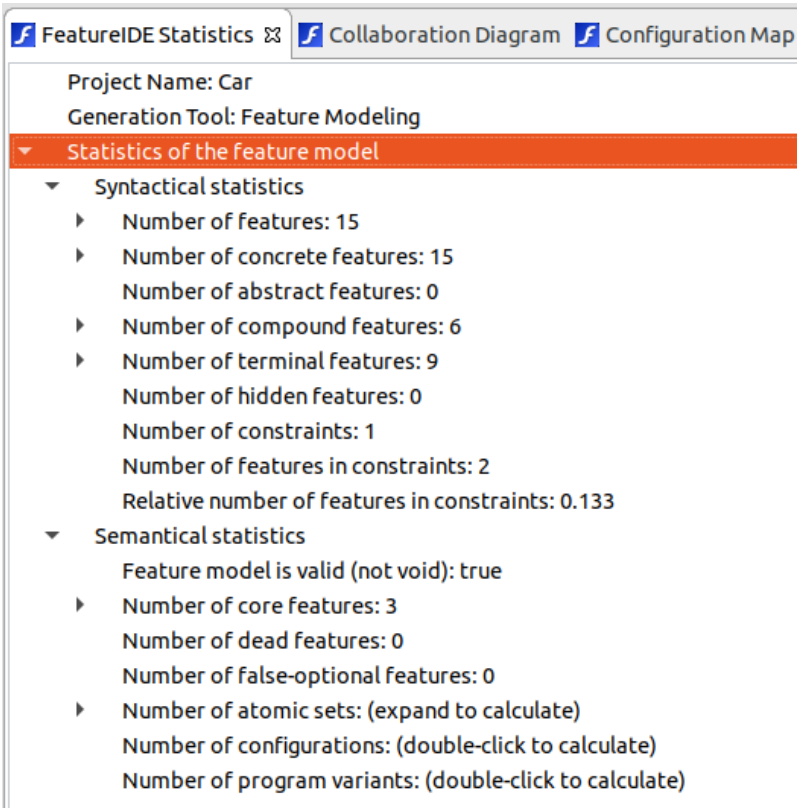

Figure 5.3: FeatureIDE Statistics View

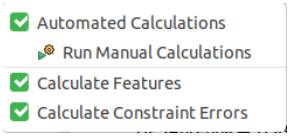

Figure 5.4: Feature Model Editor Analyses Dropdown

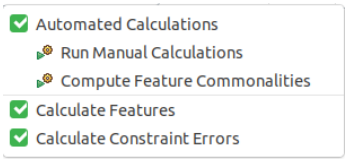

Figure 5.5: Feature Model Editor Analyses Dropdown With Commonality

https://doi.org/10.24355/dbbs.084-202009161329-0

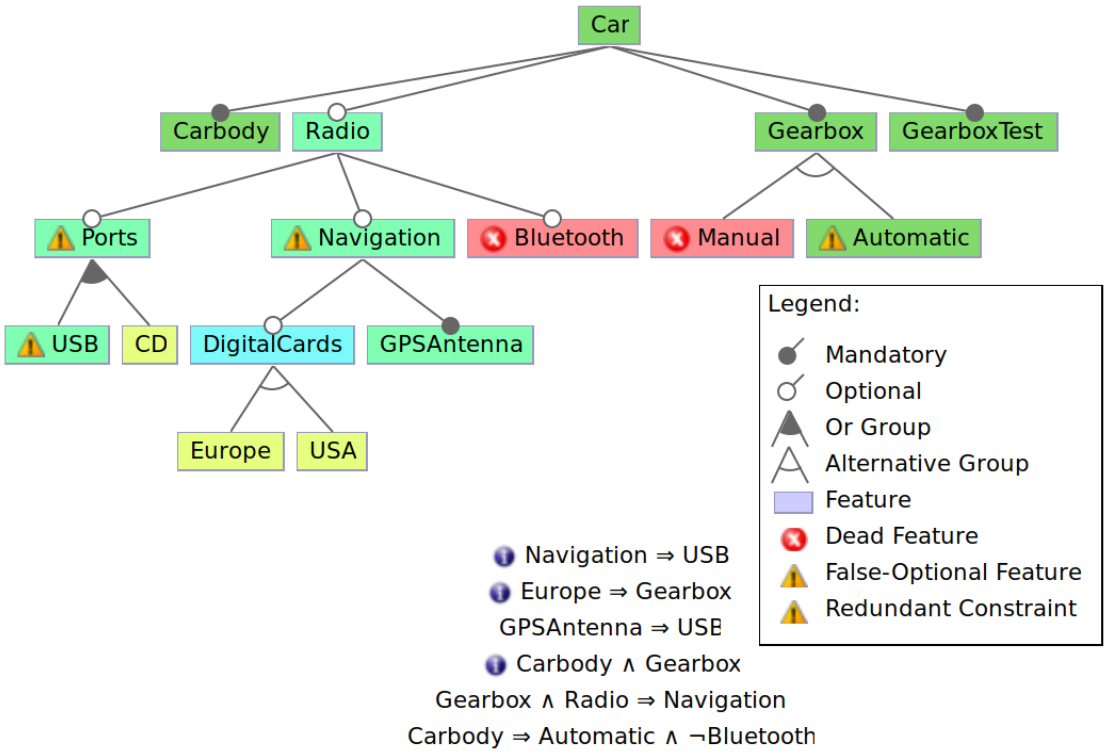

Figure 5.6: Features Colored by their Commonalities

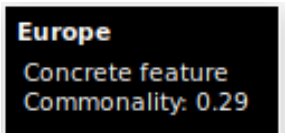

Figure 5.7: Feature Tooltip with Commonality

https://doi.org/10.24355/dbbs.084-202009161329-0

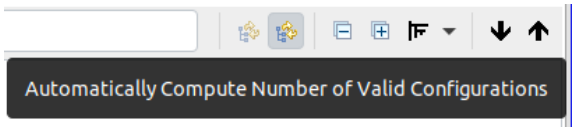

Figure 5.8: Partial Configuration Count Toggle

(a) Toggle Off (b) Toggle On

Figure 5.9: Partial Configuration Count

the current version of FeatureIDE, the previously described procedure using blocking clauses is performed to compute the number of valid configurations. After a short timeout, the number of valid configurations that have been found during this time is displayed. Thus, a lower bound is provided. However, these lower bounds are typically very imprecise for feature models with a high number of valid configurations. After a feature has been selected this number is recomputed.

We replace this computation with an algorithm that is based on d-DNNFs. As this computation is expensive, this can be prevented by a newly introduced toggle that is shown in Figure 5.8. In this case, the editor only displays whether the current partial configurations is valid or invalid as seen in 5.9(a). If the toggle is enabled the number of remaining configurations is computed using a d-DNNF based approach as seen in 5.9(b).

When the user enables the toggle, a d-DNNF that is equivalent to the feature model is computed and stored. Then, after each change to the configuration editor a query is run on the stored d-DNNF. This query computes the number of remaining valid configurations of the current partial configuration. This allows for faster online queries after the initial expensive computation of the d-DNNF.

Integration Uniform Random Sampling

FeatureIDE provides a wizard that can be used to generate configurations using different algorithms. Currently, it is possible to generate configurations by the following strategies: configurations that cover t-wise interactions, randomly but not uniformly distributed configurations, all valid configurations, or all current configurations from the **configs**-folder. After selecting such a strategy the algorithm can be selected. Each algorithm computes a set of literal sets that represent the computed configurations. These literal sets are then used to create configuration files in the directory **products**.

We added the d-DNNF-based procedure to perform uniform random sampling as a possible algorithm for the random strategy to this wizard which is shown in Figure 5.10. FeatureIDE provides an abstract class `ARandomConfigurationGenerator` that can be used to implement a generator for random samples. Using this class, we only had to implement our procedure described in Section 5.1 in the interface method `generate()` that creates a list of sets of literals which each correspond to one configuration.

https://doi.org/10.24355/dbbs.084-202009161329-0

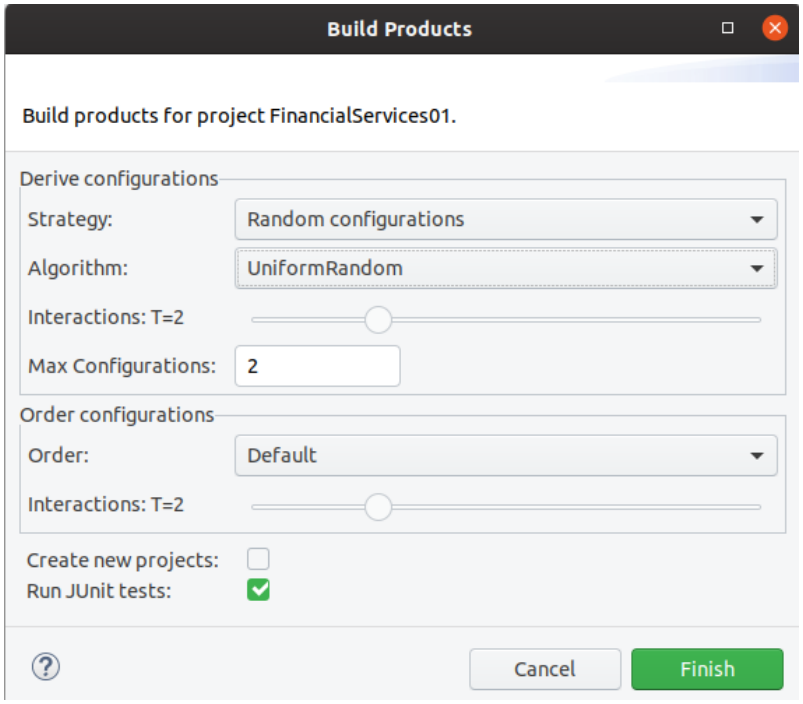

Figure 5.10: Example of the Product Generator Wizard

5.3 Evaluation Framework

In this section, we describe the implementation of our benchmark framework we use for the empirical evaluation. The implementation is focused on enabling easy addition of new #SAT solvers and new algorithms dependent on #SAT. The framework is implemented in Java. This section explains the structure of the framework, helps readers to repeat the experiments, and supports them in adding their own solvers and algorithms for further comparisons. We uploaded our benchmark framework to a Github repository.[4]

Structure

The two main parts of the benchmark framework are the different algorithms and #SAT solvers. Wrappers are required to call and parse the result of the #SAT binaries with our framework to use them in Java to allow further usage of the results for the algorithms implemented in Java. For each solver, an instance of `ICompa-rableSolver` needs to be implemented. A class diagram is shown in Figure 5.11. `executeSolver(dimacsPath, timeout)` handles the execution of the solver and passes the required arguments to the solver. The returned `BinaryResult` will indicate a timeout should one occur, otherwise it holds the `stdout` stream of the solver. `parseResult(stdout)` parses the stream to extract the computed number of satisfying assignments. The returned `SolverResult` will indicate a failure should one occur, otherwise it holds the number of satisfying assignments. We define methods to retrieve useful descriptors. `getIdentifier()` provides a unique identifier for the solver. The solvers can be grouped by different types which can be arbitrary strings.

[4] https://github.com/SundermannC/sharpsat-benchmark-framework

https://doi.org/10.24355/dbbs.084-202009161329-0

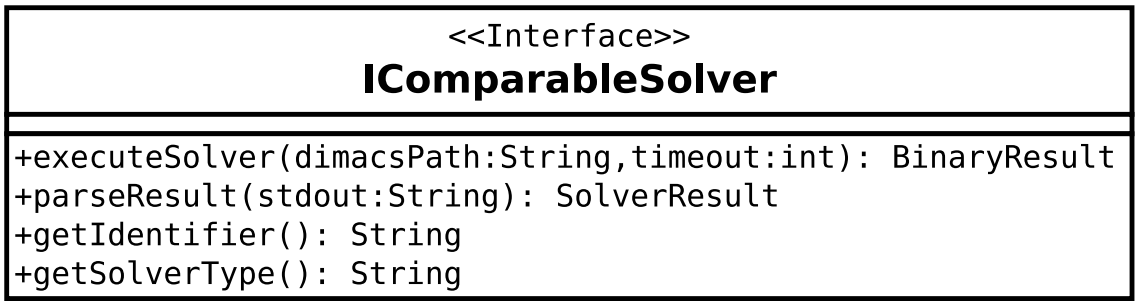

Figure 5.11: Class Diagram for d-DNNF Structure

`getSolverType()` can be used to return the type of the solver. An example for a solver type used in the framework is `"DPLL"`.

The framework enables to compare multiple algorithms on the same task, as we aim to identify the best performing algorithms and optimizations for specific task. For example, an experiment may compare two different algorithms that perform uniform random sampling. To this end, two different instances of `IComparableAlgorithm` that perform the same computation can be implemented. A class diagram for `IComparableAlgorithm` is shown in Figure 5.12. The method `measureRuntime(file,solver,timeout)` is used measure the runtime and memory usage of a single algorithm which can be used to compare multiple algorithms of the same type (e.g., algorithms that compute commonalities). The returned `Map<String,String>` can be used to return arbitrary algorithm outputs (e.g., a configuration created by uniform random sampling).

The framework also allows to directly compare the performance of solvers when used for the same algorithm, as we aim to identify the best performing solver for a specific task. `compareSolvers(file, solvers, timeout)` runs the entire algorithm once but the required #SAT calls are repeated once for every solver that was passed with `solvers`. The returned `CompareSolverResultPackage` contains the computed number of satisfying assignments, the required runtime, and used memory for each solver for each #SAT call required for the algorithm.

For some algorithms, it may not be obvious to detect computationally expensive parts in the implementation. `preciseAnalysis(file, solvers, timeout)` allows to measure the runtime of specific parts of an algorithm. The method returns a `PreciseAnalysisResultPackage` object which provides runtimes required for specific parts that can be specified by the user. Additionally, it provides the number of performed #SAT calls.

Usage

In this section, we describe how to use the benchmark framework. First, we describe how to repeat the experiments that are used for our empirical evaluation. Second, we explain how to configure new experiments using the existing solvers and algorithms. Third, we describe how to add new solvers and algorithms to the benchmark framework.

The main method lies in `src/main/RunBenchmark`. It expects the name of a configuration file as argument. If it is not provided as call argument, the user is asked to

https://doi.org/10.24355/dbbs.084-202009161329-0

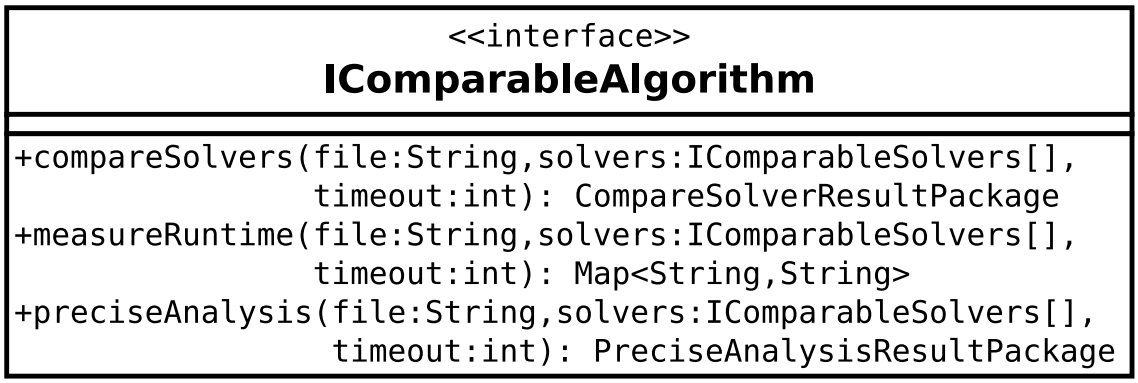

Figure 5.12: Class Diagram for d-DNNF Structure

specify a file during runtime. A Configuration file needs to be stored in the `configs`-directory. Every configuration used for the evaluation is stored in this directory in the repository. For example, to repeat `experiment1`, **"java src/main/RunBenchmark experiment1"** can be used.

To create their own experiment, the user must understand the elements of the experiment configurations. The configuration an unordered key-value store. A template for the configuration is shown in Listing 5.17.

```
1 type:<compareAlgorithms|compareSolvers|preciseAnalysis>
2 solvers:<all|type|[solver1,...,solvern]>
3 files:<all|[dir1,...,dirn]>
4 seed:w
5 timelimit:x
6 memoryLimit:y
7 incrementSize:z
8 timeUnit:<m|s|ms|ns>
```

Listing 5.17: Experiment Configuration Template

A mandatory argument is the **type** of the benchmark which can be either *compareAlgorithms*, *compareSolvers*, or *preciseAnalysis* which correspond to the three different methods of `IComparableAlgorithm` explained above. The selection of solvers can be done using the **solvers** key. The legal values are either *all*, a type (e.g., *DPLL*), or a list of n solvers *[solver1,...,solvern]*. A single solver can be selected using *[solver]*. Without the brackets, the parser tries to find a solver type (e.g., DPLL) named *solver* instead. The default argument for **solvers** is *all*. The evaluated feature models can be selected with the **files** key which only considers files that are stored in the directory `models/`. The possible arguments are either *all* or a list of m sub-directories *[dir1,...,dirn]* of `models/`. Like for **solvers**, *all* is the default argument.

We also expose variables which can be set to control the behavior of the benchmark. Some experiments are dependent on random number generation (e.g., uniform random sampling). Therefore, it is possible to provide a seed for reproducible results with the key **seed**. Furthermore, a timeout in minutes can be set using **timelimit**. If this value is not specified, there is no timeout. Additionally, the RAM usage of the solvers can be limited using **memorylimit** whose input is interpreted as MB. As the computations using #SAT are often computationally expensive and

https://doi.org/10.24355/dbbs.084-202009161329-0

require a lot of time, it is possible run the experiments incrementally. To this end, intermediate results are stored after every x processed models. If the experiment is interrupted, the experiment can be continued from the last saved state. The size x of the increments can be specified using **incrementsize**. Depending on the expected runtime of the experiments the results that show the required runtime may be more readable using different time units. This can be specified with **timeunit**. The possible values are either m (minutes), s (seconds), ms (milliseconds), or ns (nanoseconds). The results for a experiment are saved in **results/<experimenttype>/<algorithmtype>/<configname>.csv** (e.g., **results/compareSolvers/commonality/experiment1.csv**).

One of the design goals for the framework is the easy addition of new #SAT solvers. To add a new solver to the framework only two steps are required. First, one needs to write a wrapper class for the solver that is an instance of `IComparableAlgorithm`. Afterwards, the wrapper needs to be registered in the class `comparablesolver/SolverProvider.java` which holds a list of all solvers and is used to select the solvers according to the experiment configuration. Listing 5.18 shows a simplified example for a wrapper for the #SAT solver `Cachet`.

```
1  public class ComparableCachet implements IComparableSolver {
2
3    private final static String ID = "cachet";
4
5    private int memoryLimit;
6
7    public ComparableCachet(int memoryLimit) {
8      this.memoryLimit = memoryLimit;
9    }
10
11   @Override
12   public BinaryResult executeSolver(BinaryRunner runner, String
         dimacsPath, long timeout) throws InterruptedException {
13     String command = buildCommand(dimacsPath);
14     BinaryResult output = runner.runBinary(command, timeout);
15     return output;
16   }
17
18
19   @Override
20   public SolverResult getResult(String output) {
21     final Pattern pattern = Pattern.compile("Number of solutions\\s
           +[0-9]*\\.?[0-9]*[eE]?[+-]?\\d+");
22     final Matcher matcher = pattern.matcher(output);
23     String result = "";
24     if (matcher.find()) {
25       result = matcher.group();
26     } else {
27       return SolverResult.getUnexpectedErrorResult();
28     }
29     final String[] split = result.split("\\s+");
30     return SolverResult.getSolvedResult(split[split.length - 1]);
31   }
32
33   @Override
34   public String getIdentifier() {
```

https://doi.org/10.24355/dbbs.084-202009161329-0

```
35      return ID;
36    }
37
38    @Override
39    public String getSolverType() {
40      return SolverTypes.DPLL;
41    }
42
43  }
```

Listing 5.18: Example Solver Wrapper Cachet

Adding a new algorithm is similar to adding a new solver. First, one needs to create an instance of `IComparableAlgorithm` that computes the desired results. Listing 5.19 shows a simplified example for an `IComparableAlgorithm` that computes the number of valid configurations. Second, the instance has to be added at `comparablealgoriths/basics/AlgorithmProvider.java`.

```
1  public class NaiveModelCount implements IComparableAlgorithm {
2    public static final String ALGORITHM_ID = "NaiveModelCount";
3
4    public static final String ALGORITHM_GROUPID = "count";
5
6    @Override
7    public CompareSolverResultPackage compareSolvers(BinaryRunner runner,
         String file,
8        List<IComparableSolver> solvers, int timeout, IPreprocessResult
             preprocessResult) throws
9        InterruptedException {
10     Map<String, List<InstanceResult>> results = new HashMap<>();
11     List<InstanceResult> resultPackage;
12     BinaryResult binaryResult = null;
13     SolverResult solverResult = null;
14     String modelName = FileUtils.getFileNameWithoutExtension(file);
15     IFeatureModel model = FMUtils.readFeatureModel(file);
16     FMUtils.saveFeatureModelAsDIMACS(model, DIMACSUtils.
           TEMPORARY_DIMACS_PATH);
17     resultPackage = new ArrayList<>();
18     for (IComparableSolver solver : solvers) {
19       long startTime = System.nanoTime();
20       binaryResult = solver.executeSolver(runner,DIMACSUtils.
             TEMPORARY_DIMACS_PATH, timeout);
21       long runtime = BenchmarkUtils.getDurationNano(startTime, System.
             nanoTime());
22       runner.killProcessesByUserAndName(solver.getBinaryName());
23       solverResult = solver.getResult(binaryResult.stdout);
24       resultPackage.add(InstanceResult.mergeBinaryAndSolverResult(
             solverResult, binaryResult, runtime, timeout));
25     }
26     results.put(modelName, resultPackage);
27     return new CompareSolverResultPackage(solvers, results,
           ALGORITHM_ID);
28   }
29
30   @Override
31   public Map<String, String> measureRuntime(BinaryRunner runner, String
         file,
```

https://doi.org/10.24355/dbbs.084-202009161329-0

```
32        IComparableSolver solver, int timeout, IPreprocessResult
              preprocessResult) throws
33        InterruptedException {
34      Map<String, String> results = new HashMap<>();
35      BinaryResult binaryResult = null;
36      SolverResult solverResult = null;
37      String modelName = FileUtils.getFileNameWithoutExtension(file);
38      IFeatureModel model = FMUtils.readFeatureModel(file);
39      FMUtils.saveFeatureModelAsDIMACS(model, DIMACSUtils.
            TEMPORARY_DIMACS_PATH);
40      binaryResult = solver.executeSolver(runner,DIMACSUtils.
            TEMPORARY_DIMACS_PATH, timeout);
41      solverResult = solver.getResult(binaryResult.stdout);
42      results.put(modelName, solverResult.result.toString());
43      return results;
44    }
45
46    @Override
47    public Map<String, String> measureRuntime(BinaryRunner runner, List<
          String> files,
48        IComparableSolver solver, int timeout, IPreprocessResult
              preprocessResult) throws
49        InterruptedException {
50      Map<String, String> results = new HashMap<>();
51      for(String file : files) {
52        Map<String, String> interimResult = measureRuntime(runner,file,
              solver, timeout, preprocessResult);
53        results.putAll(interimResult);
54      }
55      return results;
56    }
57
58    @Override
59    @SuppressWarnings("unused")
60    public PreciseAnalysisResultPackage preciseAnalysis(BinaryRunner
          runner, String file,
61        IComparableSolver solver, int timeout, IPreprocessResult
              preprocessResult) throws
62        InterruptedException {
63      PreciseAnalysisResultPackage results = new
            PreciseAnalysisResultPackage(file);
64      results.startClock("Init");
65      BinaryResult binaryResult = null;
66      SolverResult solverResult = null;
67      String modelName = FileUtils.getFileNameWithoutExtension(file);
68      results.stopClock("Init");
69      IFeatureModel model = FMUtils.readFeatureModel(file);
70      FMUtils.saveFeatureModelAsDIMACS(model, DIMACSUtils.
            TEMPORARY_DIMACS_PATH);
71      results.startClock("solver");
72      binaryResult = solver.executeSolver(runner, DIMACSUtils.
            TEMPORARY_DIMACS_PATH, timeout);
73      results.stopClock("solver");
74      solverResult = solver.getResult(binaryResult.stdout);
75      results.maxMemory = maxMemory;
76      results.maxMemorySource = maxMemorySource;
77      return results;
```

https://doi.org/10.24355/dbbs.084-202009161329-0

```
78    }
79
80    @Override
81    public String getAlgorithmId() {
82      return ALGORITHM_ID;
83    }
84
85    @Override
86    public String getAlgorithmGroupId() {
87      return ALGORITHM_GROUPID;
88    }
89
90  }
```

Listing 5.19: Example Algorithm Wrapper

The three methods `measureRuntime()`, `compareSolvers()`, and `preciseAnalysis()` typically share a majority of the code. Copying the code for each method is error-prone and expensive to update. Thus, we create a code generator that uses a source file that needs to contain the algorithm once with some tags which we describe later added to it. The idea is to implement the procedure once without worrying about source code that is used for the evaluation of the algorithm. Also, updating an algorithm would otherwise require to update the three methods separately. Given the created procedure with some tags added to it, the generator creates the actual wrapper with the three methods `measureRuntime()`, `compareSolvers()`, and `preciseAnalysis()`. In the following, we present this generator in short. Listing 5.20 shows the annotated source code that was used to generate Listing 5.19 with the help of our generator.

```
1  public class NaiveModelCount {
2
3    public void measureRuntime(String file, IComparableSolver solver, int
         timeout) {
4
5      //#RTB:Init
6      BinaryResult binaryResult = null;
7      SolverResult solverResult = null;
8      String modelName = FileUtils.getFileNameWithoutExtension(file);
9      //#RTE:Init
10
11     //#RT:ReadModel
12     IFeatureModel model = FMUtils.readFeatureModel(file);
13
14     //#RT:SaveDimacs
15     FMUtils.saveFeatureModelAsDIMACS(model, DIMACSUtils.
           TEMPORARY_DIMACS_PATH);
16
17     //#CSOL:modelName
18
19     //#RESULT:modelName;solverResult.result.toString()
20
21     //#RETURN
22   }
23 }
```

Listing 5.20: Example Solver Wrapper Cachet

https://doi.org/10.24355/dbbs.084-202009161329-0

Tag	Explanation
#CSOL	Invoke the solver.
#RESULT	Save an intermediate result.
#RETURN	Return the overall result.
#RT	Measure the required time for the following statement. Only used for `preciseAnalysis`.
#RTB	Starts clock for a code block. Only used for `preciseAnalysis`.
#RTE	Stops clock for code block and saves measured time. Only used for `preciseAnalysis`.

Table 5.2: Generator Annotations

The generator copies regular program lines for each method and replaces annotations differently depending on which of three of three methods `measureRuntime()`, `compareSolvers()`, and `preciseAnalysis()` is currently built. Table 5.2 shows an overview of the important available tags. The general syntax for an annotations is //#<**TAG**>:<**arg1**>;<**arg2**>. For every method **#CSOL:<resultname>** is replaced with a statement that invokes a solver. For `compareSolvers()`, every passed solver is invoked once successively and the required runtime, memory, and result is saved for each. For `preciseAnalysis()`, the runtime is measured and the counter that saves the number of #SAT calls is incremented. **#RESULT:<key>:<result>** is ignored by `compareSolvers()` and `preciseAnalysis()`. For `compareAlgorithms()`, it is replaced with a statement that saves **result** in the `Map<String,String>` that is returned in the end. **#RETURN** is replaced with a statement that returns the result of the analysis as `CompareSolverResultPackage`, `PreciseAnalysisResultPackage`, and `Map<String,String>`. The tag **#RT:<key>** is ignored by `compareSolvers()` and `compareAlgorithms()`. For, `preciseAnalysis()` it is replaced with statements that measure the runtime of the following statement. **#RTB:<key>** an **#RTE:<key>** behave similarly but are used to measure the time of an entire code black instead of a single statement. **#RTB** needs to be put at the beginning and **#RTE** end of the block.

To add a model to the framework, an user must add the model.xml file to the `models/` directory. Currently, the framework only supports formats that are also supported by FeatureIDE. Thus, every model needs to be translated to a FeatureIDE format before it can be used in the framework.

5.4 Summary

We first describe the implementation of algorithms for the four different analyses, namely computing the number of valid configurations of a feature model, compute the commonality of a feature, compute the number of remaining valid configurations of a valid configurations, and perform uniform random sampling. First, we describe the general procedure that is used for the algorithms that contains the following steps: **(1)** read and parse the feature model, **(2)** translate the feature model to CNF, **(3)** adapt the formula according to the query, **(4)** invoke the solver with the adapted formula as input, and **(5)** parse the output of the solver. Afterwards, we

https://doi.org/10.24355/dbbs.084-202009161329-0

describe the required adaptations to the CNF and different optimizations for the different analyses. We explain the implementation for our d-DNNF engine that can be used for all the listed analyses. The engine utilizes a off-the-shelf d-DNNF compiler to translate the CNF to d-DNNF. Then, we store the resulting d-DNNF in our own format which is used for the different queries.

We added our d-DNNF engine and four analyses using it to FeatureIDE. First, we replaced the computation for the number of valid configurations for a feature model in the view that displays statistics of the feature model. Second, the commonality of a feature can be visually displayed in the editor for feature models. The commonality is indicated by coloring the different features and presented in a tool-tip. Third, we replaced the computation for the remaining valid configurations of a partial configurations. Fourth, we added uniform random sampling as an algorithm for the product generator that was already part of FeatureIDE.

In the third section, we describe a benchmark framework that can be used to compare #SAT solvers on analyses for feature models. The main design goals are to enable simple integration of new #SAT solvers, algorithms, and feature models. Experiments can be parameterized with a key-value store configuration file which can be used to specify the used solvers, algorithms, models, and properties as a timeout or memory limit. We use the described benchmark framework for our empirical evaluation which is described in the following chapter.

https://doi.org/10.24355/dbbs.084-202009161329-0

6. Evaluation

In this chapter, we examine the scalability of the implementations presented in Chapter 5. For the evaluation, we consider 12 exact and 2 approximate off-the-shelves #SAT solvers and 131 industrial feature models. The results are used to answer the research questions *RQ4-RQ6* introduced in Chapter 1. Overall, we aim to identify effective optimizations, solvers with short runtimes, and the scalability of the applications especially regarding large industrial feature models.

For each of the four analyses considered in Chapter 4, we identify the best performing algorithm and discuss their scalability for industrial feature models. We also discuss the efficiency of d-DNNFs as a reasoning engine for counting-based analyses of feature models. In addition, we examine the performance of #SAT solvers for the different algorithms. We provide insights in the benefits of using an approximate #SAT solver to estimate the number of valid configurations.

In Section 6.1, we describe the research questions that we answer with our empirical evaluation in more detail. In Section 6.2, we present the underlying experiment design. Hereby, we specify the technical setup, give an overview of the evaluated implementations, describe the different #SAT solvers, and discuss the product lines included in our benchmark. In Section 6.3, we present the results of the experiments. In Section 6.4, we discuss the results of our experiments separated and ordered by the research questions *RQ3-RQ5*. In Section 6.5, we discuss potential threats to the validity of our experiments.

6.1 Research Questions

In this section, we further specify the research questions *RQ3-5* introduced in Chapter 1 that we aim to answer with our empirical evaluation. We provide a survey for *RQ1* and *RQ2* in Chapter 3.

- *RQ3: For a given #SAT application, is there an algorithm that scales to industrial product lines?* In Chapter 3, we show that each considered #SAT

https://doi.org/10.24355/dbbs.084-202009161329-0

application can be computed by one of the following analyses: the number of valid configurations of a feature model, the commonality of features, and the number of remaining valid configurations of a partial configuration. While uniform random sampling is also an application that is dependent on partial configurations, we decided to separate it due to its relevancy and optimizations specifically for uniform random sampling. It is sufficient to show these four analyses scale to industrial product lines as the results can be used to compute the results for each considered #SAT application. Therefore, we evaluate the algorithms described in Section 5.1 on the feature models presented in Section 6.2.1 with the off-the-shelf solvers discussed in Section 6.2.2. We separate the research question according to the specific analyses.

  - *RQ3.1: For computing the number of valid configurations of a feature model, is there an algorithm that scales to industrial product lines?*
  - *RQ3.2: For computing the commonalities of features, is there an algorithm that scales to industrial product lines?*
  - *RQ3.3: For computing the number of remaining valid configurations of a partial configuration, is there an algorithm that scales to industrial product lines?*
  - *RQ3.4: For uniform random sampling, is there an algorithm that scales to industrial product lines?*

- *RQ4: For a given algorithm, is one #SAT solver superior to the others?* We expect that there are #SAT solvers that are overall faster than other solvers. We aim to examine whether a solver is faster for a specific algorithm but slower for others. Such conclusions may help a developer to select a solver depending on the used algorithm.

- *RQ5: What is the performance of approximate #SAT solvers for analyzing product lines?* For some #SAT applications, the exact number of valid configurations is not necessary and it may be beneficial to estimate results for faster runtimes. We evaluate the runtime of two approximate #SAT solvers on evaluating the number of valid configurations to examine whether the usage of them yields benefits.

6.2 Experiment Design

In this section, we specify the technical setup and the evaluated algorithms, product lines and #SAT solvers. The implementation of the benchmark framework is discussed in Chapter 5. To find a solution, the applications presented in Chapter 3 all require either computing the number of valid configurations of a product line, the commonality of features, the number of remaining valid configurations of a partial configuration, or uniform random sampling. Thus, we evaluate different implementations for each of those four computations. These implementations consist of the naive base algorithms and at least one optimization presented in Chapter 4. In the following, we describe all performed experiments during our evaluation. We provide

https://doi.org/10.24355/dbbs.084-202009161329-0

i	Title	Algorithms	RQs	Solvers	Timeout
1	Feature Models	Model Counting	*RQ3.1, 4*	All	5 min.
2	Commonality	Naive, Propagate Analyses, d-DNNF	*RQ3.2, 4*	Remaining	10 min.
3	Partial Configurations	Naive, d-DNNF	*RQ3.3, 4*	Remaining	10 min.
4	Uniform Random Sampling	Naive, Propagate Analyses, d-DNNF	*RQ3.4, 4*	Remaining	15 min.
5	Approximates	Model Counting	*RQ5*	Approximates	5 min.

Table 6.1: Overview Experiments

an overview in Table 6.1. For each experiment, we first explain the contribution to our research questions. Second, we specify the technical details. The memory usage is limited to eight gigabytes for the JVM and each solver. Overall, if solvers reach the timeout for a specific algorithm, it is impossible to differentiate how much time the solvers require to compute actual result. Therefore, we aimed to maximize the timeouts for the different experiments while finishing all experiments within the timeframe of this thesis. For this goal, we decided to set the timeout to ten minutes as a baseline for the experiments. To put this in perspective: a single solver which always hits the timeout of 10 minutes for 131 models requires 22 hours for a single algorithm, this translates to almost 2 weeks of total time to evaluate all 14 solvers on that algorithm. Increasing the timeout would allow us to conduct fewer experiments in our evaluation. For some experiments, we decided to change the timeout. In each of these cases, we discuss our reasons for this change.

In the *first experiment*, one goal is to identify solvers to exclude from following experiments for performance reasons. A single solver that always hits the timeout vastly increases the runtime required for the experiment. Furthermore, computing the number of valid configurations of a feature model is the threshold problem for every evaluated algorithm. Every solver that is slow for that task will be slow for every other algorithm as well. For example, to compute commonalities of a feature model with 1,000 features 1,000 #SAT calls are required for the base algorithm. We expect that each of those calls requires a similar amount of time to counting the number of valid configurations once. Thus, a #SAT solver that does not scale for model counting should not scale for the other applications, namely computing commonalities, computing remaining valid configurations of a partial configuration, and performing uniform random sampling. Additionally, the results are used to provide insight on the following sub-research questions: *RQ3.1* and *RQ4*. We set a timeout of five minutes for each single model. Each solver that does not compute the number of valid configurations for at least 10% of the systems is excluded from the following experiments. We expect that analyses in subsequent experiments to require a multiple of the runtime for this experiment, as such we reduced the timeout by a factor of two. Therefore, we do not consider a solver that reaches a timeout of

https://doi.org/10.24355/dbbs.084-202009161329-0

five minutes on a majority of the models as suitable for the following experiments. In this chapter, we use the term *remaining solvers* to identify solvers which fulfilled these requirements.

In the *second experiment*, we examine the scalability of computing the commonality of all features for a given feature model. To this end, we evaluate the three different implementations for computing the commonality of features described in Chapter 5. We aim to answer the sub-research questions *RQ3.2* and *RQ4*. For the experiment, we consider all remaining solvers and all industrial models. However, there is a technical limitation with the algorithm that inputs the d-DNNF engine as it can only be evaluated with the two d-DNNF compilers that produce a smooth d-DNNF, namely `C2D` and `dSharp`. `d4` does not support creating smooth d-DNNFs. In the remainder of this chapter, we refer to an algorithm that does employ knowledge compilation (i.e., uses a d-DNNF) as direct computation. The timeout is set to 10 minutes for computing the commonality of every feature for each solver and algorithm.

In the *third experiment*, we examine the scalability of computing the number of remaining valid configurations of a partial configuration. To this end, we evaluate the two different implementations considered in Chapter 5. We aim to provide insight on the research questions *3.3* and *RQ4*. For the experiment, we consider the remaining solvers and all considered feature models. The implementations need to compute the number of remaining valid configurations of 200 different partial configurations for each model. These consist of 50 of each with 2, 5, 10, 50 randomly included or excluded features. A considered partial configuration is not necessarily valid. The timeout is set to 10 minutes. During this time, 200 $(50 * 4)$ partial configurations of a single feature model need to be evaluated.

In the *fourth experiment*, we examine the scalability of performing uniform random sampling for a given feature model. To this end, we evaluate the three different implementations for performing uniform random sampling described in Chapter 5. We aim to answer the research questions *RQ3.4* and *RQ4*. The implementations need to create 10 uniform random configuration. The timeout is set to 15 minutes for the following two reasons: First, creating samples is typically not used interactively for the user and samples can be created in the background with a longer runtime. Second, we expect uniform random sampling to be more expensive than the other analyses. During the 15 minutes, 10 configurations of a single feature model need to be created.

In the *fifth experiment*, we examine the scalability of approximate #SAT solvers on computing the number of valid configurations for industrial feature models. We argue that these results for the scalability of approximate #SAT solvers can be transferred to the other applications (e.g., computing commonality), as the formulas only differ in a minority of the clauses. We aim to answer the research question *RQ5*. For this experiment, we only consider the two approximate #SAT solvers and all feature models described in the following section. We set the timeout to five minutes for the same reasons as for experiment one. During this time, the evaluated approximate #SAT solver needs to estimate the number of valid configurations for a single feature model.

https://doi.org/10.24355/dbbs.084-202009161329-0

Technical Setup

In this section, we describe the technical setup of our experiments. Hereby, we specify the properties of the host system and the JVM.

The experiments were run on a machine with a *Linux Centos 7* operating system and a *64-bit* architecture. The machine has an *Intel Core Broadwell Processor* consisting of *16* sockets which each has one core. The clock rate of the processor is *2,394.47 Mhz*. Overall, the machine contains *62 GB* of RAM.

The compilation of our benchmark framework described in Section 5.3 to a .jar has been performed with the Java Development Kit version *1.8.0_252*. The Java Runtime Environment version used to run the experiments was *1.8.0_232-b09*. The following parameters have been set: *-d64 -Xmx8g* which limited the memory usage of the Java Virtual Machine to 8GB. The runtimes were measured using Java's `System.nanoTime()` at the start and the end of an algorithm's implementation.

6.2.1 Subject Systems

We argue that the scalability of #SAT dependent applications on industrial product lines is the most relevant aspect. If the applications do not scale for industrial models, they are currently not usable for the industry. Thus, we only considered industrial product lines in constrast to synthesized ones. Overall, we used product lines from the automotive, operating system, database, and financial services domain. An overview of the used product lines is provided in Table 6.2. Some product lines are grouped for readability, such as CDL and KConfig. Here, #Models corresponds to the number of different product lines from that group. For multiple product lines, an evolution that contains several version of the feature model is available to us. In this case, we always consider the latest version.

Subject Systems	#Models	#Features	#Constraints
KConfig	7	96–6,467	14–3,545
CDL	116	1,178–1,408	816–956
Automotive01	1	2,513	2,833
Automotive02	1	18,616	1,369
Automotive03	1	588	1184
Automotive04	1	531	623
Automotive05	1	1,663	10,321
FinancialServices	1	771	1,080
BusyBox	1	631	681
BerkeleyDB	1	76	20

Table 6.2: Overview Subject Systems

The majority of product lines, namely the ones translated from KConfig, translated from CDL, and Automotive02 are provided by a benchmark from Knüppel et al. [KTM+17]. The models can be found here.[1] KConfig is a tool used to manage configurable systems that was originally developed for Linux [OGB+19]. CDL was

[1] https://github.com/AlexanderKnueppel/is-there-a-mismatch

https://doi.org/10.24355/dbbs.084-202009161329-0

developed for the eCos system [VD10]. Knüppel et al. [KTM+17] translated the described models and four snapshots of an automotive product line provided by their industry partner to FeatureIDE [MTS+17] format. For our evaluation, we use the models in the FeatureIDE format.

In our previous work [STS20], we introduced Automotive03-05 which represent three different automotive product lines provided by our industry partner. We translated the three product lines from a proprietary format to FeatureIDE format.

Automotive01, FinancialServices, and BerkeleyDB are available as example feature models in FeatureIDE and are based on industrial product lines. FeatureIDE examples are uploaded in the project's repository.[2] BusyBox is a software product line also specified in KConfig and is available in another Github repository.[3]

6.2.2 #SAT Solvers

In this section, we present the #SAT solvers used for the evaluation. Overall, we evaluated 12 exact and 2 approximate #SAT solvers. Table 6.3 gives an overview of the exact #SAT solvers considered in the experiments. We only considered solvers that are publicly available and accept DIMACS as input format. Each of the exact solvers is either based on DPLL or is a knowledge compiler (i.e., compiles the original input formula to another format that allows faster counting) which is indicated by the "Target Format"-column in Table 6.3.

Solver	Type	Target Format	Reference
PicoSAT	DPLL	-	[Bie08]
Relsat	DPLL	-	[BJP00]
sharpCDCL	DPLL	-	[SBB+04, SBK05a]
Cachet	DPLL	-	[KMM13]
sharpSAT	DPLL	-	[Thu06]
countAntom	DPLL	-	[BSB15]
C2D	Compiler	d-DNNF	[Dar02, Dar04]
dSharp	Compiler	d-DNNF	[MMBH10]
d4	Compiler	d-DNNF	[LM17]
miniC2D	Compiler	SDD	[OD15]
CNF2OBDD	Compiler	OBDD	[TS16]
CNF2EADT	Compiler	EADT	[KLMT13]

Table 6.3: Overview Exact #SAT Solvers

`PicoSAT` [Bie08] is a SAT solver that is not intended to be used for counting the number of satisfying assignments. However, it supports enumerating all satisfying assignments and it allows to suppress printing them all out. Thus, it can be used to compute the number of solutions. The options we used to enable model counting are: **-n** and –**all**. For our empirical evaluation, we used the `PicoSAT` release 965.[4]

[2] https://github.com/FeatureIDE/FeatureIDE
[3] https://github.com/PettTo/Measuring-Stability-of-Configuration-Sampling
[4] http://fmv.jku.at/picosat/

https://doi.org/10.24355/dbbs.084-202009161329-0

`Relsat` [BJP00] is a DPLL-based #SAT solver. It optimizes DPLL adapted for counting by decomposing the formula in sub-formula during the DPLL procedure. After a variable is assigned and boolean constraint propagation is performed, the algorithm tries to decompose the remaining problem into independent subproblems. For our empirical evaluation, we used the `Relsat` version v2.02.[5]

`Cachet` [SBB+04, SBK05a] is a DPLL-based #SAT solver. First, the solver exploits component caching. After a sub-problem has been solved by the procedure, the result is stored for re-use. Thus, the sub-problem only needs to be solved once. Second, it uses clause learning. If the DPLL procedure finds an unsatisfying assignment, the reason for the failure is stored in form of a clause. Both techniques have already been used for regular SAT but are more promising for #SAT as it is more likely to require to solve the same sub-problem multiple times [SBB+04]. For our empirical evaluation, we used the `Cachet` version v1.21.[6]

`SharpCDCL` [KMM13] is a DPLL-based #SAT solver. The tool iteratively finds solutions and conjuncts a negation of the solution to the CNF to count the number of satisfying assignments. This concept is called blocking clauses and we it introduced in Chapter 2. For our empirical evaluation, we used `SharpCDCL` version v2.2.[7]

`SharpSAT` [Thu06] is a DPLL-based #SAT solver. It uses the optimizations that were used by `Relsat` and `Cachet`, namely component decomposition, clause learning, and component caching. In addition, Thurley et al [Thu06] proposed to not store clauses with one or zero unassigned literals to reduce required space, as these are handled by boolean constraint propagation anyways. For our empirical evaluation, we used version v13.02.[8]

`CountAntom` [BSB15] is a DPLL-based #SAT solver. It allows multi-threading. To this end, the authors proposed a caching procedure that does not fault on an unexpected order of the traversed nodes. Otherwise, parallel computation may compute incorrect results if a thread visits certain nodes too early. Thus, the caching procedures `cachet` and `sharpSAT` cannot be directly applied for multi-threading. During the entire empirical evaluation, we evaluated `CountAntom` with four threads. For our empirical evaluation, we used `CountAntom` version v1.0.[9]

`C2D` [Dar02, Dar04] compiles a propositional formula in CNF to d-DNNF containing Or-nodes and decision nodes. The procedure creates a binary decomposition tree whose leaves are the clauses of the CNF at hand. If two child clauses of a node share no common variables, the node is already in d-DNNF. Otherwise, shared variables are eliminated from the clauses with a case analysis on the variable. Consider two clauses c_1, c_2 that share exactly one variable $v \in c_1, c_2$. Let $c_i^v, c_i^{\neg v}$ be the clause where v has been eliminated from c_i by setting v to $\top, \bot$, respectively. Then $c_1 \wedge c_2 \iff (v \wedge c_1^v \wedge c_2^v) \vee (\neg v \wedge c_1^{\neg v} \wedge c_2^{\neg v})$ holds. Furthermore, the new expression fulfills determinism and decomposability. The resulting d-DNNF is saved in the format described in Chapter 5. The compiler can either be used solely as a model

[5] https://code.google.com/archive/p/relsat/
[6] https://www.cs.rochester.edu/u/kautz/Cachet/
[7] http://tools.computational-logic.org/content/sharpCDCL.php
[8] https://github.com/marcthurley/sharpSAT
[9] https://projects.informatik.uni-freiburg.de/projects/countantom

https://doi.org/10.24355/dbbs.084-202009161329-0

counter. In this case, the d-DNNF is not saved afterwards. `C2D` also supports smoothing the d-DNNF. This is recommended for model counting. For our empirical evaluation, we used `C2D` version v2.20.[10]

`dSharp` [MMBH10] compiles a propositional formula in CNF to d-DNNF. The compilation is based on `sharpSAT`. The formula is also decomposed disjoint components like for `C2D`, but this is done dynamically during the procedure. The d-DNNF is saved in the format described in Chapter 5. The compiler can be used solely as a model counter. In this case, the d-DNNF is not saved afterwards. `dSharp` also supports smoothing the d-DNNF. For our empirical evaluation, we used the `dSharp` version with the commit tag `b8b252`.[11]

`d4` [LM17] compiles a propositional formula in CNF to d-DNNF. D4 also uses a dynamic decomposition like `dSharp`. The d-DNNF is saved in the format described in Chapter 5. The compiler can either be used solely as a model counter. In this case, the d-DNNF is not saved afterwards. `d4` does not support smoothing. Therefore, the compiler is not suitable for our d-DNNF based algorithms and was solely used as a model counter. For our empirical evaluation, we used the `d4` binary as the source code is not publicly available with version v1.0.[12]

`MiniC2D` [OD15] offers the compilation of a CNF to a sentential decision diagram. Additionally, it supports model counting without knowledge compilation. For our experiments, we only used `MiniC2D` as a model counter. For our empirical evaluation, we used `MiniC2D` version v1.0.0.[13]

`CNF2OBDD` [TS16] compiles a propositional formula in CNF to an OBDD. Afterwards, the paths of the OBDD that represent a satisfying assignment are counted to acquire number of solutions. For our empirical evaluation, we used `CNF2OBDD` version v1.0.2.[14]

`CNF2EADT` [KLMT13] compiles a propositional formula in CNF to extended affine decision trees. An affine clause is a XOR with a finite number of variables. An affine decision network is DAG whose leaves are terminals (i.e. $\top$ or $\bot$) and the internal nodes are either $\wedge$, $\vee$ or an affine decision node which consists of an affine clause and a left and a right child. An decision network DN is considered an extended affine decision tree, if the following properties hold: First, children of $\wedge$ or $\vee$ nodes share no variables. Second, at most one child of an affine decision node n shares variables with n. This format supports model counting in polynomial time. For our empirical evaluation, we used `CNF2EADT` version v1.0.[15]

In addition to the twelve exact #SAT solvers, we evaluated two approximate #SAT solver to answer *RQ4*. For both solvers, we explain the idea to approximate the number of satisfying assignments and describe the way we used the solver for our empirical evaluation.

[10] http://reasoning.cs.ucla.edu/c2d/
[11] https://github.com/QuMuLab/dsharp
[12] http://www.cril.univ-artois.fr/kc/d4.html
[13] http://reasoning.cs.ucla.edu/minic2d/
[14] www.sd.is.uec.ac.jp/toda/code/cnf2obdd.html
[15] http://www.cril.univ-artois.fr/kc/eadt.html

https://doi.org/10.24355/dbbs.084-202009161329-0

`ApproxMC` [CMV16, CMV13] is an approximate #SAT solver. Their idea revolves around XOR constraints that each approximately halve the space of satisfying assignment. Each variable $v_1, ..., v_n \in vars_F$ is added to the XOR constraint at a 50% chance. Without any loss of generality, let $v_i, ..., v_{i+k}$ be the k selected variables. The resulting constraint $v_i \oplus v_{i+1} \oplus ... \oplus v_{i+k-1} \oplus v_{i+k}$ is satisfied if an odd number of the k variables is $\top$. This is the case for approximately half of the valid assignments. This logic can be repeated to split the configuration space in approximately halve multiple times [BG19]. `ApproxMC` provides bounds for the confidence δ and tolerance ϵ of for the approximated result. This means that the result for the formula F lies in the interval $[(1+\epsilon)^{-1} * \#F, (1+\epsilon) * \#F]$ with a probability $p \geq 1-\delta$ [CMV13]. The developers strongly recommend to compute an independent set of variables prior to the computation of the solver. For that task they provide a tool to compute minimal independent sets. The computed set need to be saved as the first line of the DIMACS prior calling `ApproxMC`. Thus, we integrated the tool in the execution of our solver. The runtime required for the solver also considers the runtime required to compute the independent set. For our empirical evaluation, we used `ApproxMC` version v3.0.[16]

`ApproxCount` [WS05] is an approximate #SAT solver. To approximate the result, the solver iteratively assigns variables. For each assignment, a sample of satisfying assignments is computed. This sample is used approximate the number of satisfying assignments that contain the variable at hand. This is used to compute a multiplier that indicates the reduction of variability of this assignment. Consider a variable x that appears in 80% of the solutions in the sample. In this case, the multiplier for x is $\frac{1}{0.8} = 1.25$. After i iterations, the exact model counter `Cachet` is invoked on the remaining formula. The developers provide a binary of `Cachet` in their project. However, their version does not support `BigNums` which is required for our evaluation. Thus, we replaced the `Cachet` binary with the version we used for our empirical evaluation. i can be specified by the user. It is also possible to specify the number of remaining variables $vars(F) - i$. During our experiments, we set the number of remaining variables to 1000 if not stated otherwise. Previous results indicated that feature models with 1000 or fewer features are typically easy to analyze [STS20]. `ApproxCount` provides no guarantee for the quality of the result. However, their empirical evaluation indicates that the solver provides good estimates in practice [WS05, CMV13]. For our empirical evaluation, we used `ApproxCount` version v1.2.[17]

For uniform random sampling, we also analyze the tool `KUS` which is the implementation of Sharma et al. [SGRM18] for their idea which we described in Algorithm 15 in Chapter 4. The tool exploits the properties of a d-DNNF to calculate a user-specified number of configurations within a single traversal of the d-DNNF. For the translation to d-DNNF, `KUS` internally uses `d4`. The tool also demands a DIMACS file as input. The authors uploaded the source code at their GitHub repository.[18] We used the currently latest commit `9f769ec`.

[16] https://github.com/meelgroup/approxmc
[17] https://www.cs.cornell.edu/ sabhar/
[18] https://github.com/meelgroup/KUS

https://doi.org/10.24355/dbbs.084-202009161329-0

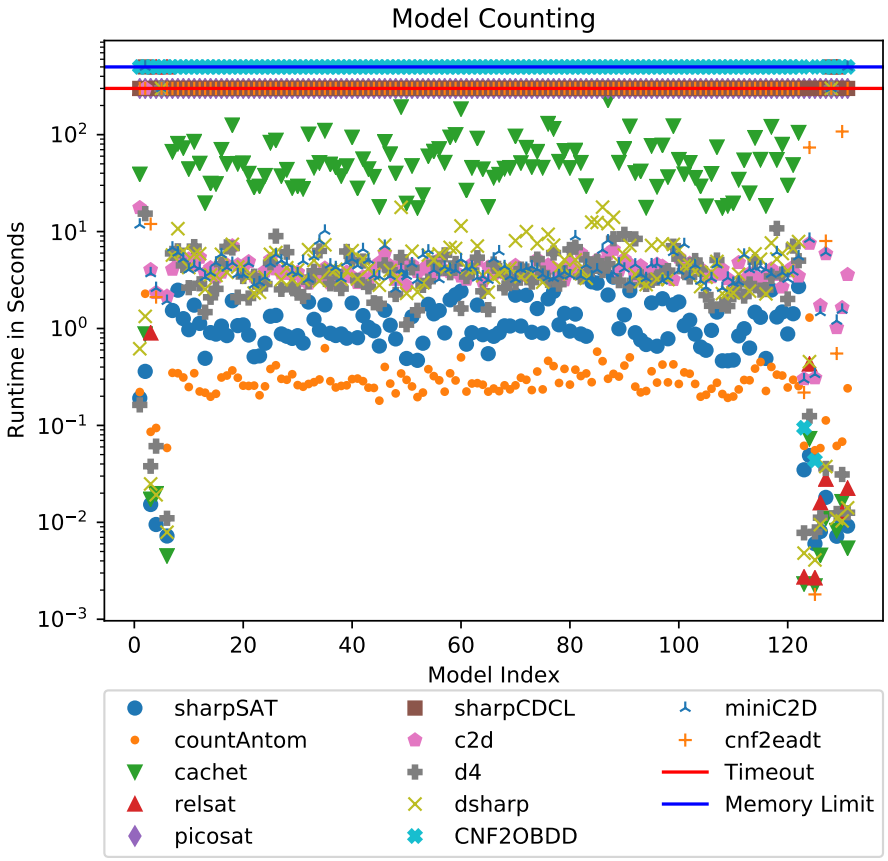

Figure 6.1: Results Experiment One

6.3 Results

In this section, we provide the results of our empirical evaluation. We separate the results according to the different experiments specified in Section 6.2.

Experiment 1: Scalability for Feature Models

Figure 6.1 shows the runtime for the twelve solvers for computing the number of valid configurations of the 131 feature models. Each point on the x-axis corresponds to one of the feature models. The models are grouped and the following order is equivalent for every diagram in the evaluation: indices **1-5** are Automotive01-05, index **6** is BusyBox, **7-122** are CDL models, **123** is BerkeleyDB, **124** is FinancialServices, and **125-131** are KConfig models. The y-axis shows the runtime of the different solvers in seconds with a logarithmic scale. The red line indicates that a solver hit the timeout of five minutes. The blue line indicates that a solver passed the memory limit of eight gigabytes. `countAntom` is the fastest solver for 116 of the 131 models and required 39.47 seconds to evaluate 129 of the 131 systems. Every solver failed to evaluate the missing two systems, namely Automotive05 and Linux. The second fastest solver is `sharpSAT` which required 150.28 seconds to evaluate 129 feature models.

Table 6.4 gives an overview for the results shown in Figure 6.1. For each evaluated solver, the table provides the absolute number of solved feature models, the percentage share of solved models, the number of times where the solver reached the timout

https://doi.org/10.24355/dbbs.084-202009161329-0

Solver	Solved	% Solved	Timeout	Memory Limit	Error
~~PicoSAT~~	0	0	131	0	0
~~Relsat~~	8	6.1	120	0	3
Cachet	125	95.4	6	0	0
~~SharpCDCL~~	0	0	129	0	2
SharpSAT	129	98.5	2	0	0
CountAntom	129	98.5	2	0	0
c2d	128	97.7	3	0	0
minic2d	127	96.9	2	0	2
dSharp	129	98.5	2	0	0
d4	129	98.5	2	0	0
~~CNF2OBDD~~	2	1.5	0	128	1
~~CNF2EADT~~	8	6.1	123	0	0

Table 6.4: Result Overview Experiment One

of five minutes, the number of times where the solver reached the memory limit, and the number of times where the solver terminated with an unexpected error. Five solvers, namely `PicoSAT`, `Relsat`, `SharpCDCL`, `CNF2OBDD`, and `CNF2EADT`, scaled to less than 7% of the feature models. `PicoSAT` and `SharpCDCL` could not even compute the number of valid configurations for a single feature model. The remaining seven solvers each successfully evaluated more than 95% of the feature models. `SharpSAT`, `CountAntom`, `dSharp`, and `d4` computed the number of valid configurations for all models, but Automotive05 and Linux. Both could not be evaluated by any of the considered solvers. The following five solvers are not considered for the experiments 2-6: `PicoSAT`, `Relsat`, `SharpCDCL`, `CNF2OBDD`, and `CNF2EADT`. These solvers are crossed out in Table 6.4.

Figure 6.2 shows the size of the d-DNNF files computed by the three d-DNNF compilers, namely `c2d`, `dSharp`, and `d4`. Each point on the x-axis corresponds to one of the 131 models. The y-axis indicates the size of d-DNNF file in kilobytes with a logarithmic scale. The d-DNNF files created by `c2d` are the smallest for 111 of the models. `dSharp` and `d4` created the smallest files for 2 and 16 models, respectively. The ranges of sizes for the d-DNNF files are: 2-21.027 kb (`c2d`), 3-152.887 kb (`dSharp`), and 1-37.253 kb (`d4`). The medians are 5.751 kb (`c2d`), 32.460 kb (`dSharp`), and 11.600kb (`d4`). The sum of the sizes for all models are 765 mB (`c2d`), 4,937 mB (`dSharp`), and 1,574 mB (`d4`).

Experiment 2: Scalability for Commonality

Figure 6.3 shows the runtime of both algorithms that use a direct computation for computing the commonality of features presented in Listing 5.2 and Listing 5.3 in Chapter 5. The diagram displays the runtime of the seven remaining exact solvers. Each point on the x-axis corresponds to one of the 131 feature models. The y-axis indicates the runtime of each solver in seconds with a logarithmic scale. The red line indicates that a solver hit the timeout. The blue lines indicates that a solver hit the memory limit. For the naive base algorithm displayed on the left side, every solver but `countAntom` reached the timeout for every CDL model. `sharpSAT`, `cachet`, `d4`,

https://doi.org/10.24355/dbbs.084-202009161329-0

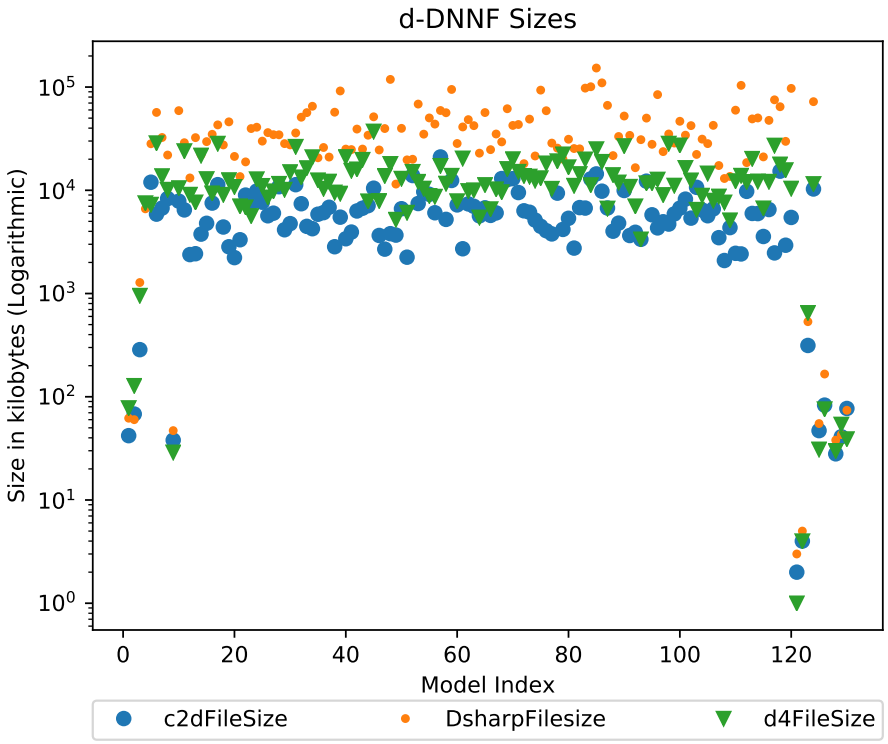

Figure 6.2: d-DNNF Sizes of `c2d`, `dSharp`, and `d4`

and `dSharp` successfully computed the commonalities for the same eleven feature models. `countAntom` successfully computed the commonalities for 119 models with an overall runtime of 14.18 hours (16.01 hours with models for which the solver hit the timeout). The right diagram shows the adaptation of the base algorithm presented in Listing 5.3 in Chapter 5. The computations for core, dead, and false-optional features passed the memory limit for two KConfig models prior to the invocation of the solver, namely Embtoolkit and Linux. In the diagram, this is denoted by a hit of the memory limit for every solver. `countAntom` successfully evaluated 127 models with an overall runtime of 7.80 hours. The other solvers successfully evaluated 32 models (`sharpSAT`), 10 (`cachet`), 11 (`d4`), 11 (`dSharp`), 7 (`c2d`), and 7 (`miniC2D`). For both algorithms based on a direct computation, every solver failed to evaluate Automotive02, Automotive05, and Linux. Overall, the best performing solver for the algorithm that included propagation of results from other analyses was faster than the best performing solver for the base algorithm for 126 models and required 7.80 hours for 128 models. The base algorithm required 15.88 hours for 128 models.

Figure 6.4 shows the runtime of the d-DNNF based algorithm with the d-DNNF compilers `c2d` and `dSharp`. `dSharp` successfully evaluated every model but Automotive05 and Linux within 4.26 hours. `c2d` additionally failed to compute the commonalities of Automotive02 which contains the highest number of features (18,616) but only required 1.50 hours to evaluate the 128 models.

Figure 6.5 shows a comparison of the runtimes for the best algorithm and solver for each the direct computation and the d-DNNF based algorithm. The best performing

https://doi.org/10.24355/dbbs.084-202009161329-0

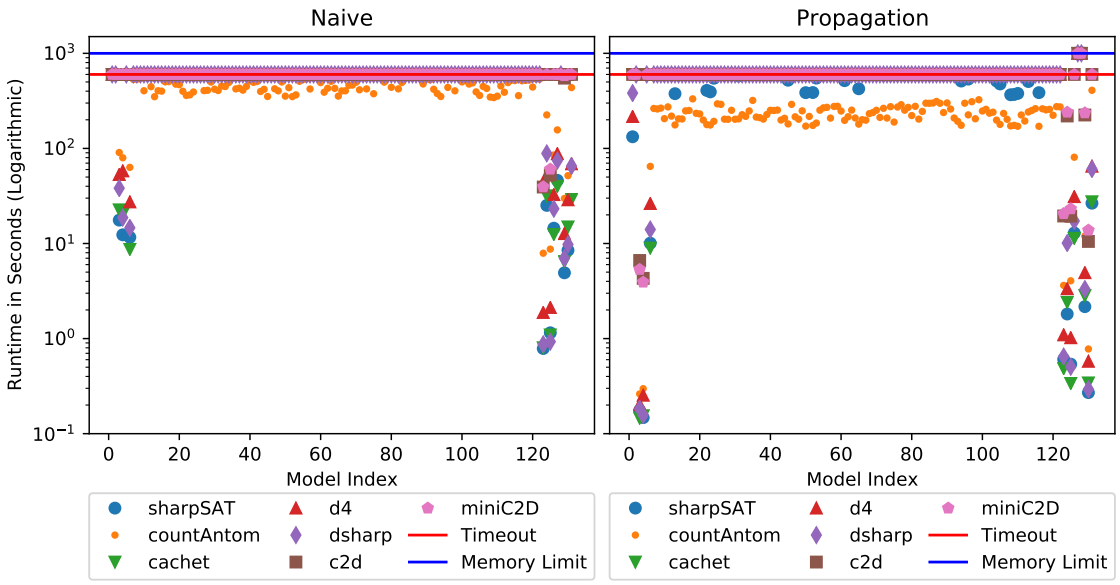

Figure 6.3: Commonality: Direct Computation

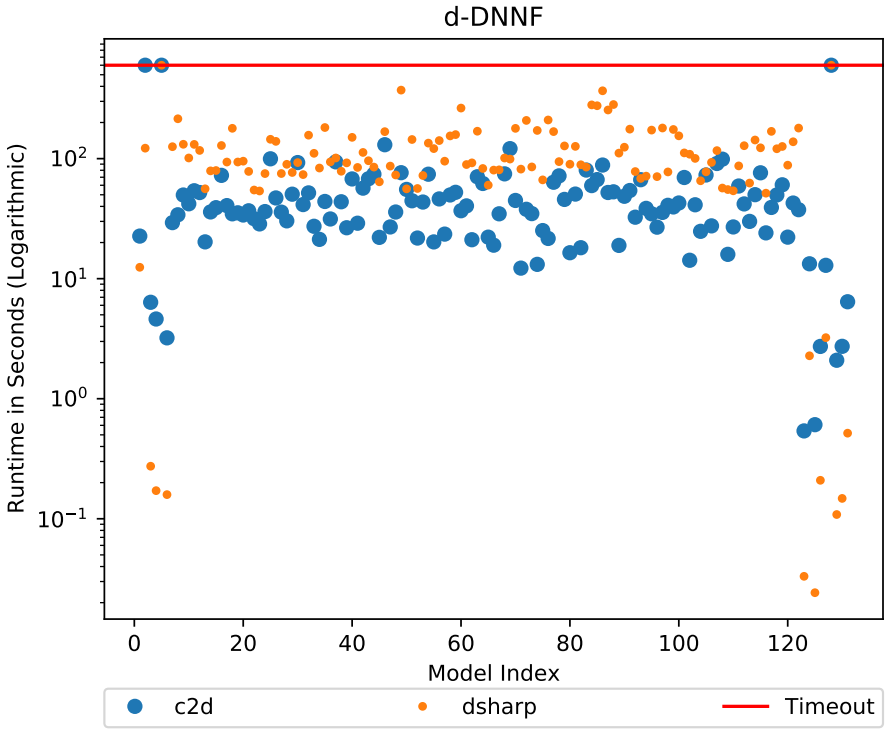

Figure 6.4: Commonality: d-DNNF

https://doi.org/10.24355/dbbs.084-202009161329-0

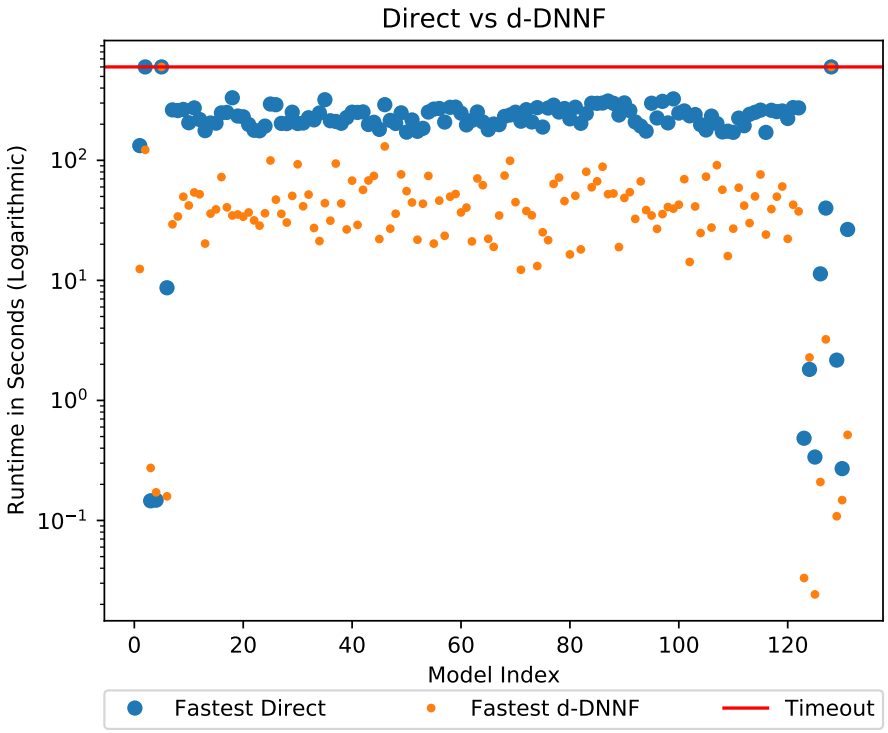

Figure 6.5: Commonality: Direct vs d-DNNF

solver for a d-DNNF is faster than the best performing solver for the direct computation for every but three models for which the d-DNNF-based algorithm required 2.72 seconds and the direct computation 2.11 seconds. In sum, the best performing solver for each model required 1.50 hours to evaluate 129 models using the d-DNNF. The best algoritihm and solver for the direct computation required 7.81 hours for 128 models.

Figure 6.6 shows the correlation between the size of the d-DNNF and the runtime of computing commonalities with our d-DNNF-based implementation for the compilers `c2d` and `dSharp`. The x-axis shows the size of the file in kilobytes with a logarithmic scale. The y-axis shows the runtime in seconds with a logarithmic scale. The red line indicates that the algorithm hit the timeout for the model. The algorithm only failed to compute the commonalities for a feature model if the compiler failed to create a d-DNNF. After the compiler successfully created a d-DNNF, the procedure did not hit the timeout for any model. For the smaller models, `c2d` and `dSharp` create d-DNNFs of similar size in which case `dSharp` has shorter runtimes.

Algorithm	#Fastest	Best Performing Solver (#Instances)
Naive	0	`countAntom`(108)
Propagation	3	`countAntom`(116)
d-DNNF	126	`c2d`(114)

Table 6.5: Result Overview Experiment Two

https://doi.org/10.24355/dbbs.084-202009161329-0

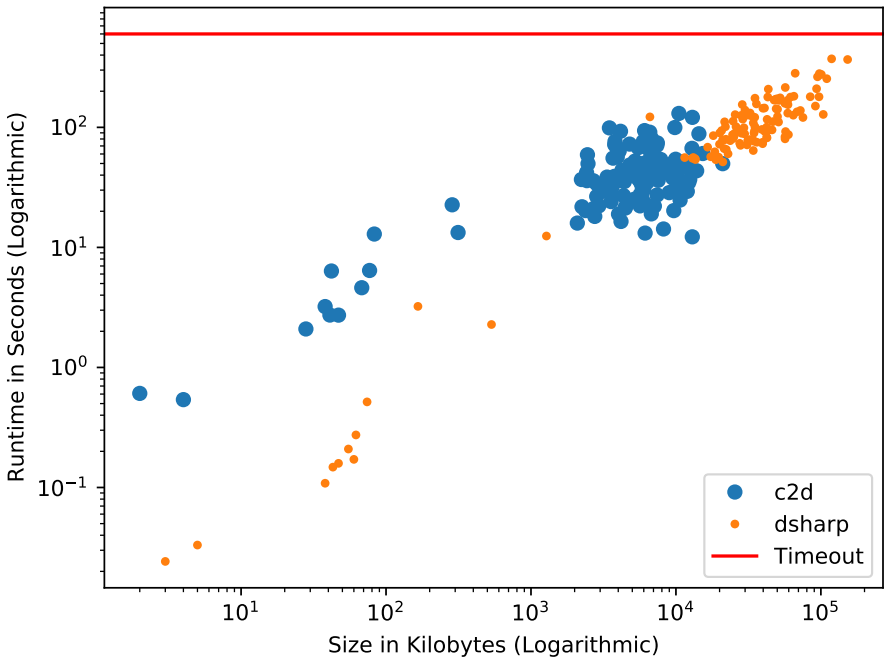

Figure 6.6: Commonality: d-DNNF Size in Relation to Runtime

Experiment 3: Scalability for Partial Configurations

Figure 6.7 shows the runtime of the remaining solvers for computing the number of remaining configurations of a partial configuration. Each point on the x-axis corresponds to one of the 131 feature models. The y-axis shows the runtime in seconds with a logarithmic scale. The solvers are differentiated by different colors and markers. The left diagram shows the results for the computations that use a direct approach where a #SAT call has been performed for every partial configuration. The right diagram shows the d-DNNF based approach where a d-DNNF was computed once per feature model and each partial configuration was evaluated with a query on the d-DNNF. Each solver had ten minutes to compute all 200 partial configurations per model.

For the direct approach, `c2d` and `cachet` evaluated five (3.82%) and seven (9.16%) feature models within ten minutes, respectively. `dSharp` evaluated all partial configurations for 115 (87.79%) feature models. `countAntom` and `sharpSAT` only reached the timeout for Automotive05 and Linux (solved 98.47%). *d4* reached the timeout on one additional (97.71%) feature model for Automotive02. `countAntom` was the fastest solver using the direct approach for 116 feature models. `sharpSAT`, `dSharp`, and `cachet` were the fastest solvers for 9, 3, and 1 feature model, respectively.

For the approach of d-DNNFs, we consider the cumulative runtime of the translation to d-DNNF and the queries for the partial configurations. Using `dSharp` the approach reached the timeout for two (solved 98.47%) feature models, namely Automotive05 and Linux. With `c2d` the approach also failed to evaluate the 100 partial configurations of Automotive02 within ten minutes (solved 97.71%). The approach with `c2d` was the fastest one in 115 cases; with `dSharp` in 14 cases.

https://doi.org/10.24355/dbbs.084-202009161329-0

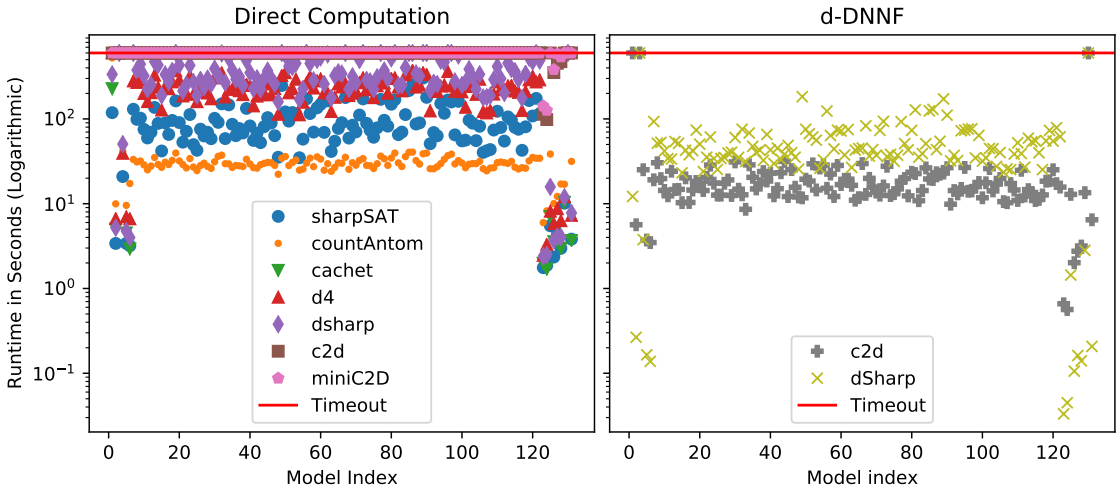

Figure 6.7: Partial Configurations: Comparison Solvers

Figure 6.8 shows a comparison of the fastest runtime using the direct and the d-DNNF-based approach. For both the direct computation and the d-DNNF approach, at least one solver computed a result for every model except for Linux and Automotive05. Overall, the d-DNNF approach required 34.61 minutes and the direct computation 61.77 minutes. The d-DNNF-based approach was faster in 126 of the 129 (97,67%) overall solved feature models. In the three instances where an algorithm using a direct computation was faster, the average difference was 1.96 seconds. Table 6.6 displays an overview of the comparison between the two techniques.

Figure 6.9 shows the correlation between the size of the d-DNNF and the runtime of evaluating partial configurations with our d-DNNF-based implementation for the compilers `c2d` and `dSharp`. The x-axis shows the size of the file in kilobytes with a logarithmic scale. The y-axis shows the runtime in seconds with a logarithmic scale. The red line indicates that the algorithm hit the timeout for the model. The correlation is similar to the results for commonality shown in Figure 6.6. There is no model for which `c2d` and `dSharp` created a d-DNNF and the algorithm hit the timeout while running the queries.

Algorithm	#Fastest	Best Performing Solver (#Instances)
Naive	3	`CountAntom`(116)
d-DNNF	126	`c2d`(115)

Table 6.6: Result Overview Experiment Three

Experiment 4: Scalability Uniform Random Sampling

Figure 6.10 displays the runtime of the remaining solvers for computing ten uniform random configurations using the base implementation Listing 5.5 (left) and Listing 5.6 (right). Each point on the x-axis corresponds to one of the 131 feature models. The y-axis shows the required runtime with a logarithmic scale. For the base algorithm, only `sharpSAT` solved any CDL feature model. Overall, `sharpSAT` successfully performed uniform random sampling for 24 feature models. No solver

https://doi.org/10.24355/dbbs.084-202009161329-0

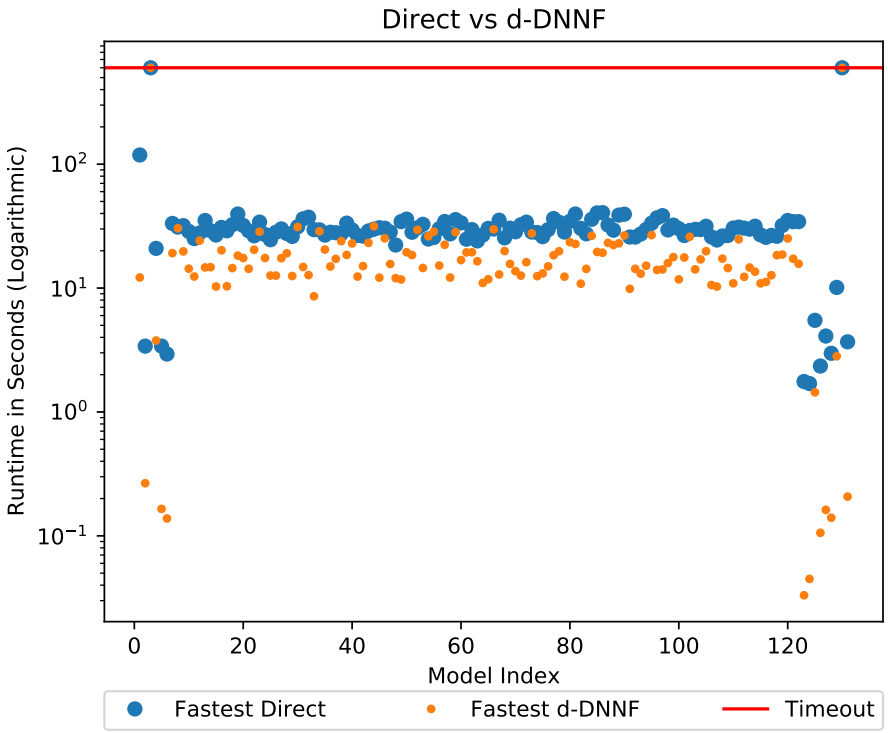

Figure 6.8: Partial Configurations: Direct vs d-DNNF

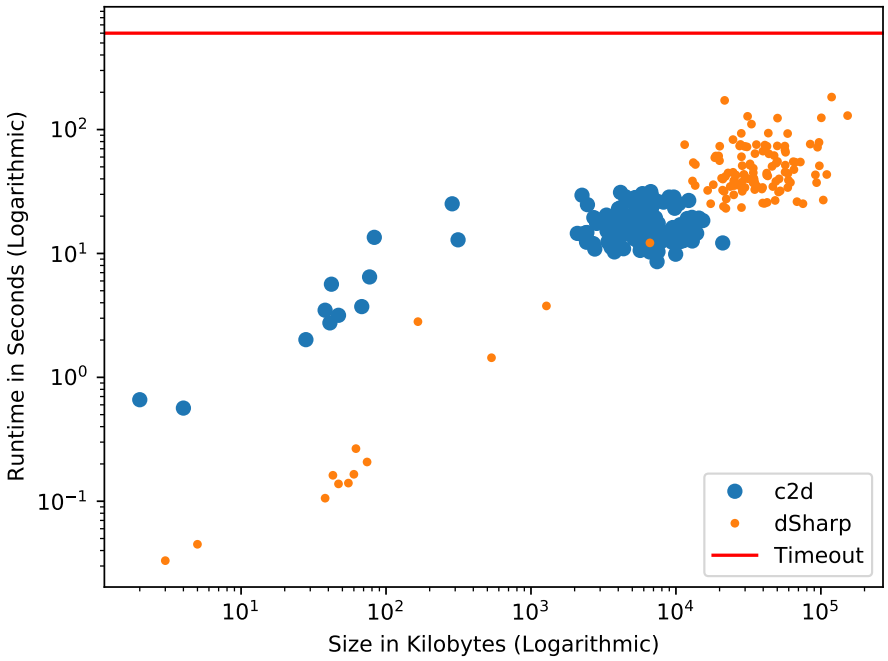

Figure 6.9: Partial Configurations: Runtime in Relation to d-DNNF Size

https://doi.org/10.24355/dbbs.084-202009161329-0

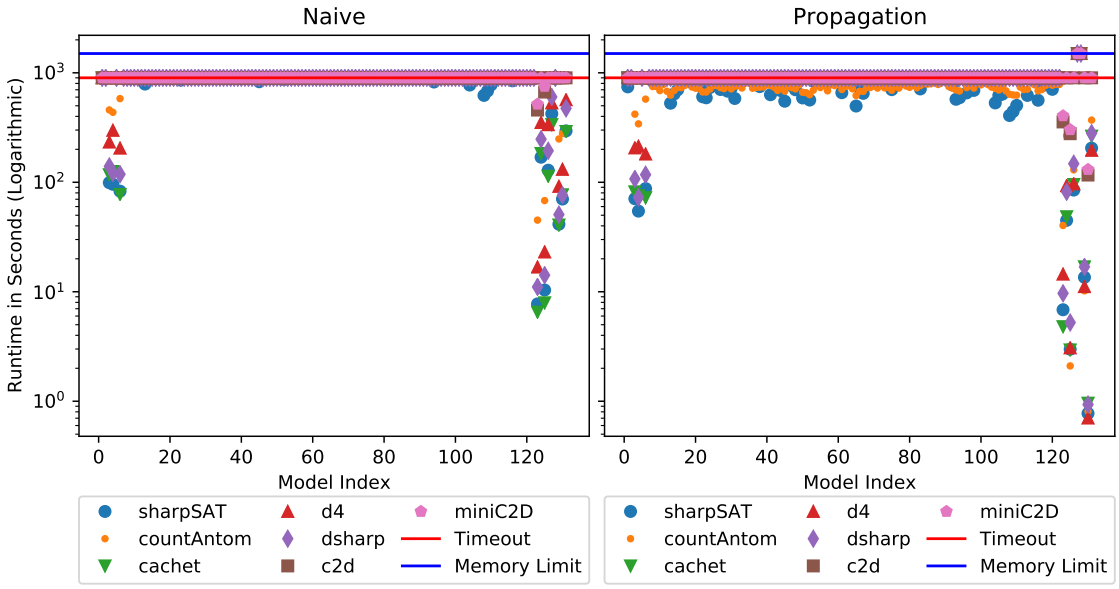

Figure 6.10: Uniform Random Sampling: Direct Computation

successfully evaluated a model outside of those 24 models. `sharpSAT` and was the fastest solver for 18 of those. `Cachet` was the fastest for the remaining six models. For the adaptation that propagates other analyses results, at least one solver successfully performed uniform random sampling for all but four models. The analyses that compute core, dead, and false-optional features reached the memory limit for Linux and Embtoolkit in the evaluation of each solver, like for commonality. In addition, each solver hit the timeout for Automotive02 and Automotive05. `countAntom` successfully evaluated 126 feature models and was the fastest solver for 87 feature models. `sharpSAT` is the only solver that evaluated a model that was not evaluated by `countAntom` (Automotive01). Overall, `sharpSAT` evaluated 60 feature models. The other five solvers evaluated at most ten feature models. `sharpSAT`, `Cachet`, and `d4` were the fastest solvers for 36, 2, and 2 feature models, respectively.

Figure 6.11 shows the runtime for computing ten uniform random samples using d-DNNFs. On the left, the runtimes with `c2d` and `dSharp` of our implementation Listing 5.16 that performs uniform random sampling by repetitively calls partial configuration queries on the created d-DNNF. On the right, the runtimes of the uniform random sampling tool `KUS` is shown. For our implementation, at least one of `c2d` and `dSharp` successfully evaluated 101 models. `c2d` was faster for 87 models and `dSharp` for 14 models. `KUS` evaluated every model but Linux and Automotive05 within the timeout of 15 minutes and required 3.72 hours for the 129 models.

Figure 6.12 displays a comparison of the fastest solver and algorithm for the two direct computations and the two computations using d-DNNFs. The fastest algorithm exploiting d-DNNFs evaluated every model but Automotive05 and Linux. For an algorithm that does not use d-DNNFs, no solver successfully evaluated Automotive02 in addition to Automotive05, and Linux. In sum, the fastest computation using d-DNNFs required 3.72 hours to evaluate the 129 models. The best performing direct computations required 24.25 hours for the 128 models.

https://doi.org/10.24355/dbbs.084-202009161329-0

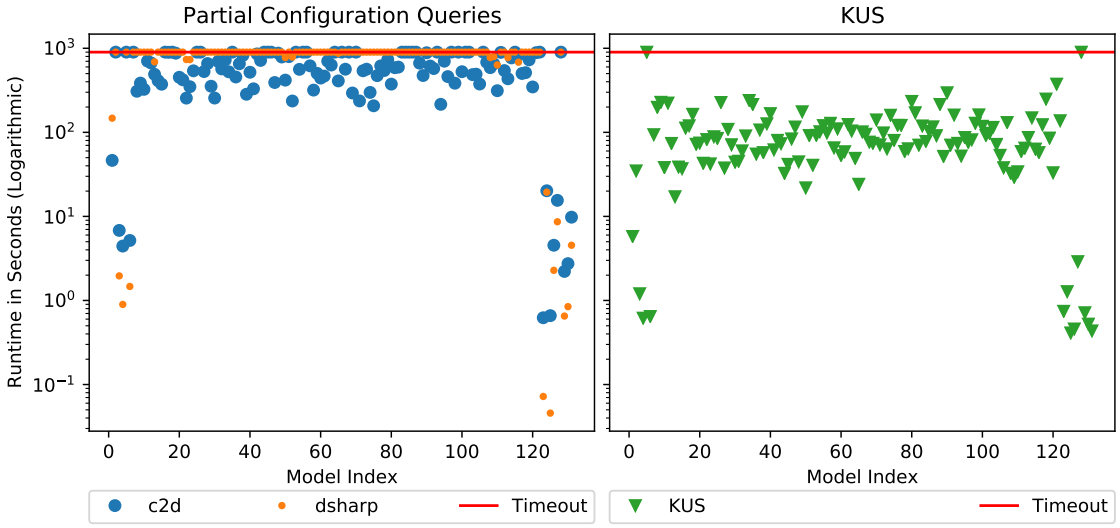

Figure 6.11: Uniform Random Sampling: d-DNNF

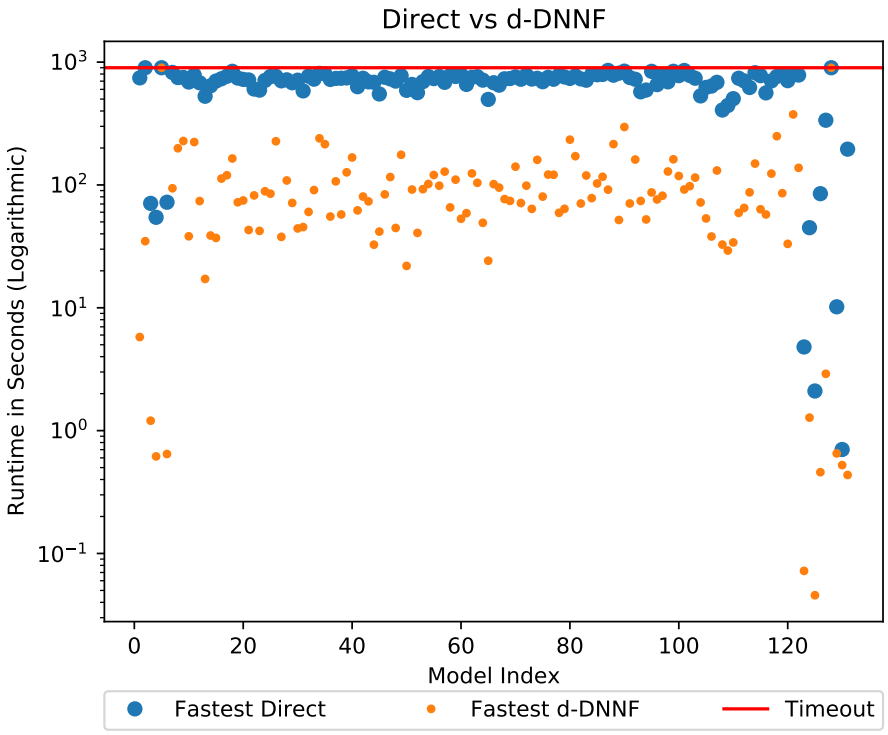

Figure 6.12: Uniform Random Sampling: Direct vs d-DNNF

https://doi.org/10.24355/dbbs.084-202009161329-0

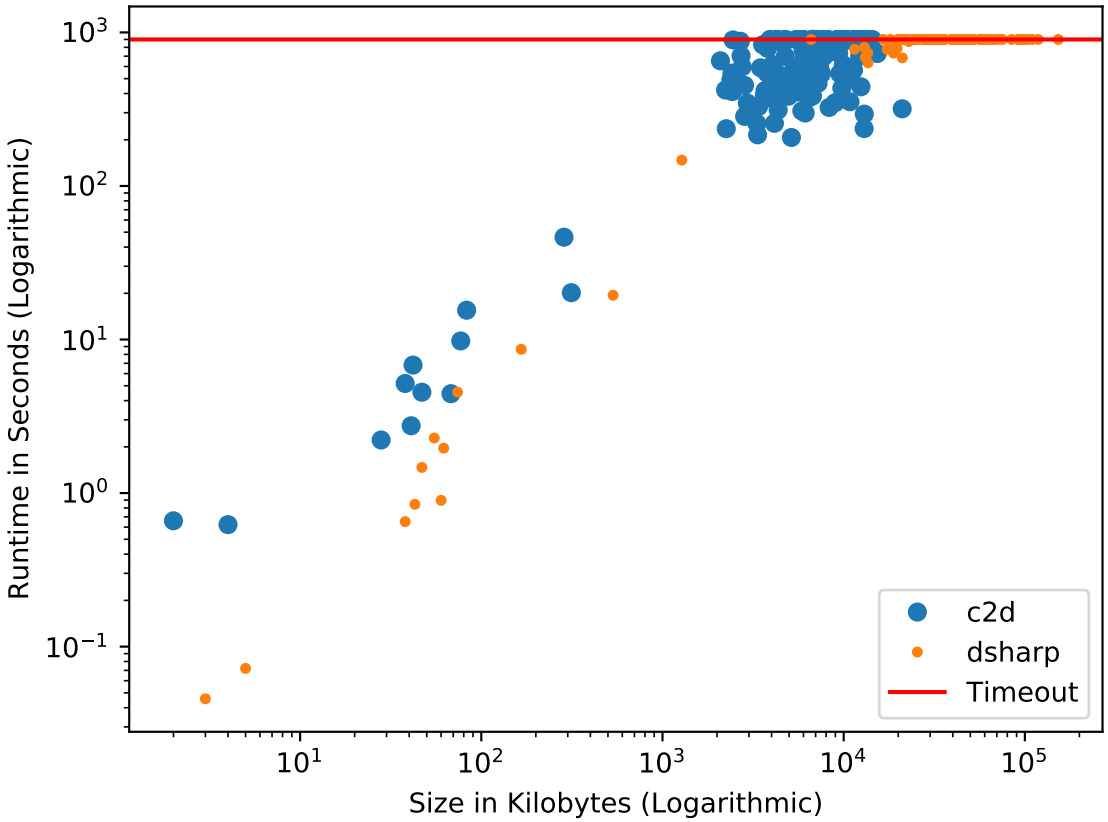

Figure 6.13: Uniform Random Sampling: Runtime in Relation to d-DNNF Size

Figure 6.13 shows the correlation between the size of the d-DNNF and the runtime of uniform random sampling with our d-DNNF based implementation using partial configuration queries for the compilers `c2d` and `dSharp`. The x-axis shows the size of the file in kilobytes with a logarithmic scale. The y-axis shows the runtime in seconds with a logarithmic scale. The red line indicates that the algorithm hit the timeout for the model. Overall, the correlation is similar to the results for commonality in Figure 6.6 and partial configurations in Figure 6.9. However, for uniform random sampling, there are models for which `c2d` or `dSharp` created a d-DNNF and the algorithm hit the timeout while running the queries. Overall, the runtime of the algorithm is higher compared to the other algorithms for our d-DNNF based algorithm. Table 6.7 gives an overview over the results.

Algorithm	#Fastest	Fastest Solver (#Instances)
Naive	0	`sharpSAT`(18)
Propagation	0	`countAntom`(87)
d-DNNF Partial Configuration Queries	3	`c2d`(87)
d-DNNF `KUS`	126	-

Table 6.7: Result Overview Experiment Four

https://doi.org/10.24355/dbbs.084-202009161329-0

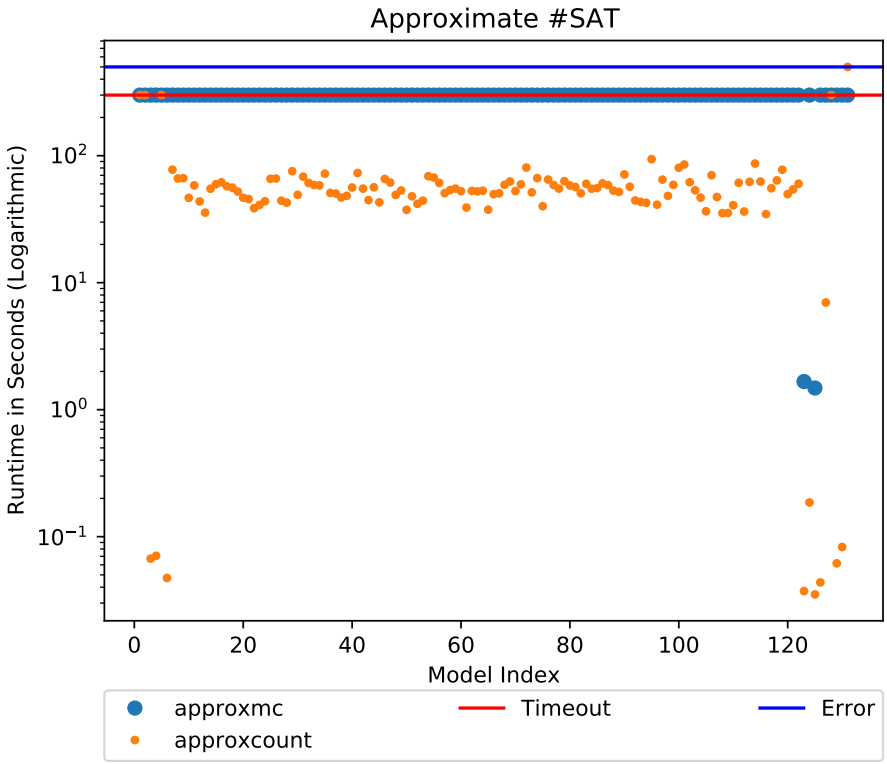

Figure 6.14: Approximate #SAT Solvers

Experiment 5: Approximate #SAT Solvers

Figure 6.14 shows the runtimes of `ApproxMC` and `ApproxCount` for estimating the number of valid configurations for the 131 feature models. Each point on the x-axis corresponds to one of the models. The y-axis indicates the required runtime with a logarithmic scale. The red line indicates that a solver hit the timeout. The blue line indicates that a solver threw an exception.

`ApproxMC` computed the result for only two feature models with 96 and 76 features respectively. For these models, the approximate #SAT solver required 1.67 and 1.48 seconds. The computed estimated numbers of satisfying assignments have an offset of 0.32% and 4.55% from the results computed by the exact #SAT solvers. For every other model, the solver reached the timeout of five minutes.

ApproxCount successfully evaluated 126 (96.18%) of the feature models. The solver hit the timeout for Automotive01, Automotive02, Automotive05, and Linux and threw an error for the KConfig model uClinux-distribution. For the 126 successfully evaluated models the solver required 2.26 hours. For comparison, `countAntom` evaluated 129 models within 39.47 seconds and, thus, was around 206 times faster. `Cachet`, which is internally used by *ApproxCount*, required 1.92 hours to evaluate 125 models on its own. For every model with fewer than 1000 features, no estimations are performed and, thus, the result is exact. There are 10 models with fewer than 1000 features in the benchmark. For every model with more than 1000 features, the number of satisfying assignments estimated by `ApproxCount` is larger

https://doi.org/10.24355/dbbs.084-202009161329-0

than the exact result. For the model with the largest difference, the estimated result is $5.72 * 10^{12}$ times larger than the exact result. The median is a factor of 165.

6.4 Discussion

In this section, we discuss the research questions *RQ3-5* using the results from our empirical evaluation. The section is separated in a part for each research question. Each part is separated in the experiments described in Section 6.2 that are supposed to partially answer the corresponding research question.

RQ3

In this section, we aim to answer *RQ3: For a given #SAT application, is there an algorithm that scales to industrial product lines?*. In Chapter 3, we argued that each of our considered applications is a use case of one of the following metrics: counting the number of valid configurations, computing the commonalities of features, counting the number of remaining valid configurations of a partial configuration, or performing uniform random sampling. Thus, we evaluated at least one algorithm for each of these metrics. Each sub-research question *RQ3.1-3.4* corresponds to one of the metrics.

The results of *experiment one* show that seven #SAT solvers, namely `Cachet`, `SharpSAT`, `countAntom`, `c2d`, `minic2d`, *dSharp*, and *d4*, evaluated more than 95% of the feature models within five minutes per model. `sharpSAT`, `countAntom`, `dSharp`, and `d4` even computed a result for every but two models, namely Linux and Automotive05. Thus, we answer *RQ4.1: Does counting the number of valid configurations scale to industrial product lines?* positively, as there are multiple #SAT solvers that compute a large majority of the product lines within seconds. This reaffirms the results of our previous work [STS20]. However, no solver scales to every feature model as Linux and Automotive05 have not been successfully evaluated by any solver.

The results of *experiment two* show using a d-DNNF query for each feature enables computing the commonalities of all features within 10 minutes for every model but Automotive05 and Linux. Thus, we answer our research question *R4.2:Does computing commonalities scale to industrial product lines?* positively. Our algorithm that exploits the properties of a d-DNNF was the fastest algorithm in 126 of the 129 evaluated cases which required 1.50 hours for the 129 models with the best performing solver for each single model. The best performing direct computation required 7.81 hours for 128 models as no solver successfully evaluated Automotive02 in addition to Automotive05 and Linux. The direction computation re-using results from core, dead, and false-optional analyses required 7.80 hours compared to the 15.88 hours required by the base algorithm. We conclude that performing those analyses is worth the effort to reduce the number of required #SAT calls. The d-DNNF based approach was 6.31 hours faster than the best performing direct computations while evaluating one additional feature model. Thus, the d-DNNF is our best performing algorithm for computing commonalities.

The results of *experiment three* show that there is an algorithm for every industrial feature model but Linux and Automotive05 that computes the number of valid configurations of 200 partial configurations within 10 minutes. Thus, we answer our

https://doi.org/10.24355/dbbs.084-202009161329-0

research question *RQ4.3: Is there an algorithm that scales to computing the number of remaining valid configurations of a partial configuration?* positively. Our algorithm that exploits the properties of a d-DNNF was faster than the direct computation for 126 of the 129 evaluated feature models, like for commonality. However, the three models for which the direct computations of commonality was faster than using d-DNNFs are different from the three models for which the direct computation for partial configuration was faster. The d-DNNF approach required 34.61 minutes and was 27.16 minutes fast than the direct computation. Thus, our d-DNNF-based algorithm performed best for computing the number of remaining valid configurations for partial configurations.

The results of *experiment four* show that there is an algorithm for every industrial feature model but Linux and Automotive05 that computes ten uniform random samples within 15 minutes. Thus, we answer our research question *RQ4.4: Is there an algorithm that scales to performing uniform random sampling?* positively. For every feature model an algorithm that exploits d-DNNFs was fastest. The tool `KUS` implemented by Sharma et al. [SGRM18] that exploits the properties of d-DNNFs was the fastest algorithm for 126 of the 129 evaluated models. Our d-DNNF implementation was the fastest for the remaining three feature models. Overall, `KUS` performed best for uniform random sampling and a d-DNNF-based algorithm was the fastest for every single model. In addition, d-DNNF are especially beneficial for uniform random sampling compared to the other three analyses which matches the expectation as uniform random sampling for ten configurations requires the highest number of queries.

For every analysis considered in the empirical evaluation, namely computing the number of valid configurations of a feature model, the commonality of features, the number of remaining valid configurations of a partial configuration, we identified an algorithm that computes a result with at least one solver. Thus, we answer our research question *RQ4: For a given #SAT application, is there an algorithm that scales to industrial product lines?* positively. For every analysis, a d-DNNF-based algorithm outperformed the algorithms that do not employ knowledge compilation. Therefore, we argue that using a d-DNNF engine for counting based analyses of feature models is promising. After computing a d-DNNF once, it can also be used for multiple different analyses which further reduces the required runtime compared to our empirical evaluation. We computed the d-DNNFs for a feature model once for every analysis. The results also indicated that there is a clear correlation between the size of the d-DNNF and the algorithms that traverse it. This matches the expectation as every d-DNNF-based algorithm we evaluated has linear runtime complexity to the number of d-DNNF nodes.

RQ4

In this section, we aim to answer *RQ5: For a given algorithm, what is the fastest off-the-shelf #SAT solver?*.

The results of *experiment one* show that `countAntom` is the fastest off-the-shelf #SAT solver for computing the number of valid configurations of a feature model for each of the 116 CDL models. For the remaining models, `countAntom` required

https://doi.org/10.24355/dbbs.084-202009161329-0

more runtime than other solvers. sharpSAT was the fastest for six models, Cachet for five, d4 for one, and CNF2EADT for one. While countAntom, sharpSAT, Cachet, MiniC2D, C2D, dSharp, and d4 all evaluated more than 95% of the feature models, PicoSAT, Relsat, SharpCDCL, CNF2OBDD, and CNF2EADT evaluated less than 10% of the models. While every d-DNNF compiler (d4, dSharp, and c2d) failed for Automotive05 and Linux, C2D also failed for Automotive02 which contains by far the most features (18,616). Of the other solvers that evaluated more than 95% of the feature models, the longest time required to evaluate Automotive02 was 15 seconds. C2D is also the only d-DNNF that does not decompose formulas dynamically but creates a decomposition tree prior to the actual compilation. It is reasonable to assume that creating the decomposition tree does not scale for feature models with a high number of features. Another observation is that c2d creates the smallest d-DNNFs regarding file size even though d4 create d-DNNFs that are not smooth which should reduce the size of a d-DNNF. Overall, countAntom is the fastest solver for computing the number of valid configurations. d4 is the fastest d-DNNF compiler and c2d creates the smallest d-DNNFs.

The results of *experiment two* show that c2d is faster than dSharp for the algorithm exploiting d-DNNFs for 114 models. dSharp is faster for 15 models. The results indicate that smaller d-DNNFs reduce the required time for the commonality queries as c2d creates smaller d-DNNFs than dSharp. For the base algorithm that uses a direct computation, countAntom is computed the commonalities of 108 CDL models. No other solver successfully evaluated a single CDL model with that algorithm. Overall, no solver evaluated a model that was not successfully evaluated by countAntom. This is similar for the adaptation which computes core, dead, and false-optional features to reduce the number of required #SAT calls. With that algorithm, countAntom evaluated every model but Automotive02, Automotive05, Linux, and Embtoolkit. The solver that successfully evaluated the second most models is sharpSAT which evaluated 24 models.

The results of *experiment three* are similar to the results for *experiment two*. c2d is faster for a majority of the models for the d-DNNF based algorithm. countAntom is the fastest solver for the direct computation for every CDL model, and, thus, for a majority of the models.

The results of *experiment four* for our algorithm using d-DNNFs are similar to the results for *experiment three* and *experiment four*. C2D is faster than dSharp for a majority of the models. For the base algorithm, sharpSAT evaluates the highest number of feature models (24). dSharp and Cachet evaluated eleven feature models. countAntom evaluated ten. For the adaptation Listing 5.6, countAntom evaluated the highest number of feature models 126 followed by sharpSAT which evaluated 60 feature models including the model Automotive01 that was not successfully evaluated by countAntom. KUS is the fastest algorithm for uniform random sampling that we evaluated. Internally, it uses d4 to compile the CNF to d-DNNF but the difference in the performance results from the time required to sample.

Overall, there is no solver that is superior to the others for all algorithms and models. However, c2d overall performs better than dsharp for all three d-DNNF based algorithms. dSharp is only faster for a few models for which both solvers required a short runtime. This makes sense as in this case the time required for

https://doi.org/10.24355/dbbs.084-202009161329-0

the translation to d-DNNF takes a larger share for the overall runtime. For the translation of smaller models, `c2d` is slower. For the algorithms based on direct computations, `countAntom` performs better overall for the majority of algorithms. However, for each analysis there are solvers which are faster for some models. For naive uniform random sampling, `sharpSAT` and `cachet` are even faster overall. The results indicate that `sharpSAT` and `cachet` are faster for queries with a short runtime for all solvers. Our naive algorithm for uniform random sampling invokes many #SAT calls with formulas that contain a high number of unit clauses which should be easier for the solvers. We assume that is the reason for the better performance for `sharpSAT` and `cachet` for that algorithm.

RQ5

In this section, we aim to answer *RQ6: How fast are approximate #SAT solvers for analyzing product lines?*. The results of *experiment five* show that both approximate #SAT solvers `ApproxCount` and `ApproxMC` are slower than the best performing exact #SAT solvers. `ApproxMC` only computed the number of valid configurations for two feature models. `ApproxCount` evaluated 125 models within the timeout of five minutes for each model. However, in sum `ApproxCount` required 1.92 hours for the 125 models while `countAntom` evaluated 129 in 39.47 seconds. There is no benefit in approximating results if the exact solvers are faster. However, we only tested the approximate #SAT solvers with one parameterization each. We assume that effective parameters result in faster runtimes but identifying effective parameterizations is beyond the scope of this thesis.

6.5 Threats to Validity

In this section, we discuss potential threats to the validity of our empirical evaluation. The threats are separated in internal and external threats.

Internal Validity

Translation to Feature Model: The translation of a configurable system to a feature model which we neglected in our thesis is a potential threat to the validity. An incorrect feature model may lead to misleading results for the applications. The majority of the models (CDL, KConfig, and Automotive02) was introduced by Knüppel et al. [KTM+17]. The authors argue that their translations possibly remove some cross-tree constraints leading to deviations to the original configurable system. We translated product lines from our industry partner in a proprietary format to create Automotive03-05 in our previous work [STS20]. The translation may also contain some errors but was reviewed several times by our industry partner. Overall, it is reasonable to assume that the evaluated models contain a few errors and simplifications compared to the original systems. However, we expect that the results for the scalability of our algorithms and the considered off-the-shelf #SAT solvers are still representative for the original systems. We considered a large variety of models overall and multiple for each domain. Thus, we assume that small errors and simplifications for single models do not result in significant changes in the scalability of the analyses.

https://doi.org/10.24355/dbbs.084-202009161329-0

Translation to CNF: In Chapter 2, we discuss that there are translations to CNF that do not preserve the number of solutions which causes incorrect results for our analyses. However, the translation provided by FeatureIDE preserves the number of solutions [MTS+17]. In addition, the performance of #SAT solvers is dependent on the translation of the propositional formula to CNF [OGB+19]. In our empirical evaluation, we only considered the translation to CNF used by FeatureIDE [MTS+17]. A different translation technique may change the performance of the evaluated #SAT calls. However, we expect that the different algorithms profit similarly for a translation that reduces the required runtime for #SAT solvers.

Computational Bias: In our empirical evaluation, we evaluated each data point only once due to the limited time and the variety of computationally expensive experiments. Thus, for a single solver, algorithm, and feature model we cannot make a statement about the internal error rate. However, we evaluated each combination of solvers and algorithm for 131 feature models. Additionally, a vast majority of these models can be separated in groups of models that are similar regarding their structure and size. We argue that using the results for the entire set of models allows representative conclusions about the performance of algorithms and solvers despite we performed the experiments only once.

JVM Warm-Up: For our empirical evaluation, we did not consider a warm-up of the JVM which potentially has an impact on the runtime of the algorithms. We decided to not include a warm-up due to the limited time. Otherwise, it would have been necessary to reduce the number of evaluated algorithms, solvers, or feature models to finish all experiments within the given time. After the evaluation, we partially repeated some of the experiments with a warm-up. None of the results indicated a relevant deviation of the measured runtimes caused by the warm-up.

Solver Invocation: We evaluated each solver with default parameterization if possible (e.g., PicoSAT requires two additional parameters to count assignments instead of proving satisfiability). Different parameters may improve the runtime of some solvers and, thus, shift the results in favor of other solvers. However, configuring the solver invocations introduces an even higher demand for computational resources and is out of scope of this thesis.

Variable Ordering: For each feature model, the ordering of the variables in the DIMACS was equal for every experiment in our evaluation. It is possible that other variable orderings change the required runtime. However, we performed some measurements with different variable orderings that indicated there is no significant difference in performance. It is reasonable to assume, that the small differences are only caused by computation bias. For a strong conclusion additional measurements are required though. We expect no significant difference as the #SAT solvers and d-DNNF compilers internally use heuristics to order variables [Thu06, SBB+04, SBK05a, BJP00, BSB15, MMBH10, LM17, Dar02].

External Validity

Comparison to Other Algorithms: In our empirical evaluation, we mainly compared own algorithms and adaptations we implemented of algorithms considered in

https://doi.org/10.24355/dbbs.084-202009161329-0

the literature. An important criterion for the quality of the algorithms is the comparison to other algorithms. The expressiveness of our results is weakened by the fact that we mainly evaluated own implementations even though we implemented them with the best intentions. We argue that our conclusions are still reasonable: (1) we identified at least one algorithm that scales to every analysis we considered, (2) we evaluated the solvers on a variety of algorithms and feature models, and (3) the d-DNNF compilers created d-DNNFs with a reasonable size for every feature model that could be evaluated by at least one #SAT solver and allows efficient queries for every considered algorithm.

Homogeneity of Models: In our empirical evaluation, the majority of feature models are based on CDL sub-systems. These are similar in size regarding number of features and number of constraints. Also, the structure of the CDL models is similar indicated by metrics such as ratio of features that appear in constraints, the average number of features that appear in a constraint, the average number of children of a feature, the tree depth, and the clause density. Thus, it is more difficult to conclude about other feature models. However, besides the CDL models we also evaluated models from multiple different domains, including operating system, automotive, and financial services. Therefore, we argue that our evaluation still allows conclusions for other feature models.

6.6 Summary

In this chapter, we described the evaluation of ten algorithms with twelve exact #SAT solvers on 131 feature models. The algorithms are used to compute results for four different analyses, namely the number of valid configurations of a feature model (one algorithm), the commonality of features (three algorithms), the number of remaining valid configurations for a partial configuration (two algorithms), and uniform random sampling (four algorithms). The results are a strong indicator that running analyses that compute core, dead, and false-optional to reduce the number of required #SAT calls is worth the effort. For every analysis, an algorithm based on the exploitation of d-DNNFs required the least runtime to analyze the models. We identified at least one algorithm that computes a result for every analysis and each feature model, except Linux and Automotive05. For both models, every solver and analysis failed. Nevertheless, we conclude that the applications we considered in Chapter 3 are applicable for industrial feature models.

Some solvers did not scale to evaluating industrial feature model, namely `PicoSAT`, `Relsat`, `SharpCDCL`, `CNF2OBDD`, and `CNF2EADT`. For every algorithm based on direct computation except the base algorithm of uniform random sampling, `countAntom` required the least runtime. For the base algorithm of uniform random sampling, `sharpSAT` had the best performance. `c2d` created the smallest d-DNNF files and performed best for each d-DNNF-based analysis implemented by us. Both evaluated approximated #SAT solvers required significantly more time to evaluate the feature models than the fastest exact #SAT solvers.

https://doi.org/10.24355/dbbs.084-202009161329-0

https://doi.org/10.24355/dbbs.084-202009161329-0

7. Related Work

In this chapter, we present work that is related to ours. First, we describe other surveys for #SAT applications. Second, we discuss works that proposed solutions for the applications we considered in this thesis. Third, we discuss the usage of knowledge compilation for the analysis of product lines. Fourth, we present alternatives to propositional logic for counting the number valid configurations. Fifth, we discuss the usage of d-DNNFs in other domains than feature-model analysis.

Surveys of #SAT Applications for Product Lines

Heradio et al. [HFACA13] performed a survey on existing work for counting the number of valid configurations and presented possible applications for estimating the benefits of product lines. The considered applications are either based on counting the number of valid configurations or the commonality of features and approximate the economic benefit of a product-line approach. Fernandez-Amoros et al. [FAGS09] also consider extracting information from product lines to estimate the benefits of a product line approach. Hereby, the authors consider the number of valid configurations, the commonality of features, and the homogeneity of a feature model. The authors also propose algorithms that are independent of off-the-shelf solvers. However, they did not implement a prototype that could be used for a comparison. In contrast to our work, the authors of both works [HFACA13, FAGS09] do not consider any application that does not estimate the economic benefits of a product-line approach vs standalone products (e.g., detecting design errors). Also, they do not provide scalable algorithms for feature models with cross-tree constraints.

Kübler et al. [KZK10], also provide a short survey of two applications of #SAT on the analysis of configurable systems. The authors consider rating errors of feature model specializations and measuring the quality of documentation via the number of valid configurations. They argue that a better product documentation leads to fewer valid configurations based on their observations with product lines of Mercedes. Overall, all surveys we are aware of discuss much fewer applications than our work and are limited in their scope (e.g., only consider economic benefit).

https://doi.org/10.24355/dbbs.084-202009161329-0

Algorithms for the Considered Applications

In this section, we discuss other works that provided solutions for the algorithms considered in this thesis, namely computing the number of valid configurations, commonality of features, remaining valid configurations of a partial configuration, or performing uniform random sampling.

Kübler et al. [KZK10] compute the number of valid configurations using #SAT for product lines from the automotive domain. They evaluated `Cachet`, `c2d`, and an own model counter which is not publicly available. Their own model counter is not based on CNFs and, thus, does not require a prior translation to CNF. The authors only evaluated the solvers for the task of computing the number of valid configurations. Additionally, their evaluation is limited to few feature models from the automotive domain that are not publicly available.

Heradio et al. [HGFACC11] propose an algorithm to compute the number of valid configurations and commonalities of features. However, they only consider feature models without cross-tree constraints. Fernandez-Amoros et al. [FAHCC14] also provide an own tool `treeCount` that is used to compute the number of valid configurations and commonalities. The tool also considers feature models with cross-tree constraints. The idea is to store all distinct partial assignments that satisfy the cross-tree constraints. The feature tree is traversed once for each partial assignment computing the resulting number of valid configurations with respect to the partial assignment and the feature tree. This was also discussed in Section 4.2. They evaluated their tool on the 30 largest models in feature model repository of `SPLOT` [1] whose largest model had 326 features at this time. Furthermore, they generated models with up-to 800 features using the `SPLOT`'s generator tool. However, these models only contained up to two-digit number of constraints. Their empirical evaluation showed that their approach performs better on these models with very few cross-tree constraints than `Cachet`. Our research for publicly available industrial models indicates that these typically contain a large number of constraints (up to 11.632 in our evaluation). In our work, we evaluated three different algorithms that use off-the-shelf #SAT solvers for computing the commonalities on these industrial models.

We are not aware of any work that specifically computes the number of remaining valid configurations of a partial configuration. However, performing uniform random sampling via repeatedly computing the number of remaining valid configurations of the current partial configuration is considered in the literature [OGB+19].

We discussed the algorithmic details of the following three approaches in Section 4.4. Oh et al. [OGB+19] proposed an algorithm for uniform random sampling using the DPLL-based #SAT solver `sharpSAT`. We also evaluated `sharpSAT` on uniform random sampling. For their empirical evaluation, they also considered KConfig models. In previous work, Oh et al. [OBMS17] use BDDs for uniform random sampling. However, the authors only evaluate the algorithm on feature models with fewer than 100 features. In following work, they argue that the approach using BDDs does not scale to KConfig systems as BusyBox [OGB+19]. Munoz

[1] http://www.splot-research.org/

https://doi.org/10.24355/dbbs.084-202009161329-0

et al. [MOP+19] also performed uniform random sampling for feature models that contain numerical features. Sharma et al. [SGRM18] use d-DNNFs to compute uniform random samples, as we do in our work. However, the used algorithm vastly differs as the authors just take a number of desired samples as input and compute all of them within a single d-DNNF traversal. We also evaluated their approach and compared it to three other algorithms on our benchmark. Their tool `KUS` is the best performing algorithm for performing uniform random sampling in our evaluation.

Knowledge Compilation in Feature Model Analysis

We only consider target languages of knowledge compilation that allow polynomial time queries for typical feature model analyses, such as the consistency of a feature model. To the best of our knowledge, d-DNNFs have only been considered for uniform random sampling in the scope of feature-model analysis [SGRM18]. Voronov et al. [VÅE11] used the similar format smooth decomposable negation normal form (DNNF) which is a super-set of d-DNNF that is not necessarily deterministic. The authors used the properties smooth and decomposable to enumerate valid configurations. They acknowledged that without the deterministic property it is not possible to compute the number of valid configurations in polynomial time. Kübler et al. [KZK10] evaluated the d-DNNF compiler `c2d` for counting the number of valid configurations. However, they solely used it as a regular #SAT solver and did not employ knowledge compilation.

Binary decision diagrams (BDDs) are widely used for the analysis of feature models [AHC+12, HPMFA+16, MWCC08, PLP11]. A variety of satisfiability-based analyses have been considered for BDDs such as checking whether a feature model is void [CW07], finding core and dead features [HPMFA+16], finding conditionally core and dead features [HPMFA+16], interactive configuration support [MWCC08], and computing the differences between two versions of a feature model [AHC+12]. Furthermore, BDDs have been used for #SAT applications as computing the number of valid configurations [BSTC07], and uniform random sampling [OBMS17]. However, BDDs grow exponentially in the worst case and multiple authors reported scalability issues regarding the size of BDDs with larger feature models [BSRC10, OBMS17, OGB+19, STS20]. The size of a BDD is heavily dependent on variable ordering and effective ordering heuristics can vastly decrease the size [MWCC08, BSRC10, HFACA13]. Even though BDDs are applicable for the analyses considered in this thesis, there is currently no tool that computes BDDs for the medium or large sized models in our benchmark.

Non-Propositional Model Counting

SAT is not the only constraint satisfaction problem (CSP) considered for the analysis of product lines [BSRC10, BTRC05, SSK+20, MOP+19]. Benavides et al. [BTRC05] used constraint programming to compute the number of valid configurations and the commonality of features. Also, they consider non-functional attributes such as a price of a feature. These can be used for additional constraints (e.g., the price of all features included in the configuration may not be higher than 50). This may further limit the set of valid configurations. However, the authors only consider feature

https://doi.org/10.24355/dbbs.084-202009161329-0

models without cross-tree constraints. Their empirical evaluation only considers very small feature models (i.e., at most 23 features).

Munoz et al. [MOP+19] also considered feature models with non-functional numerical features. They compared constraint programming, satisfiability modulo theories (SMT), and #SAT for counting the number of valid configurations and uniform random sampling. To encode the numerical features for #SAT, the authors used bit-blasting which translates an arithmetic formula into a propositional formula. The resulting formula can then be used by a SAT or #SAT solver. The authors compared sharpSAT [Thu06] (#SAT), z3 [DMB08] (SMT), and Clafer [JSRM+18] which internally uses the constraint programming library Choco [JRL08]. The empirical results show that sharpSAT with bit-blasting vastly outperformed z3 and Clafer for model counting and uniform random sampling [MOP+19]. This is interesting for potential future work that expands the algorithms we considered for non-functional attributes such as the price for feature.

d-DNNF Exploitation

In this section, we discuss usages of d-DNNFs in domains other than configurable systems. As we discuss d-DNNF compilers in Section 6.2, we exclude them here. The three tools `c2d` [Dar02], `dSharp` [MMBH10], and `d4` [LM17] take a propositional formula in CNF and translate it to d-DNNF.

Darwiche [Dar01b] proposed several algorithms for d-DNNFs. One of them is counting the number of models under assumptions which exploits the same properties as our algorithm for computing the number of remaining valid configurations for a partial configurations. Furthermore, they propose an algorithm that allows counting the number of satisfying assignments when changing a single literal regarding the current assumption without requiring a new traversal. This requires two initial traversals for each assumption set to create partial derivatives. These can be used to change the value of a single variable without re-computing the entire computation graph. We argue that this technique can be used to compute commonality of features. However, the authors provide no implementation or a technical description on how to implement their idea. To the best of our knowledge, their idea has not been empirically evaluated yet. Nevertheless, evaluating their idea for computing the commonality of features is interesting for potential future work.

https://doi.org/10.24355/dbbs.084-202009161329-0

8. Thesis Summary

Applying #SAT for the analysis of feature models enables many possible applications. In this thesis, we presented a survey which consists of applications already considered literature and our own proposals. Each application is dependent on either computing the number of valid configurations (1) of a feature model, (2) that contain a specific feature (i.e., the commonality of that feature), or (3) remaining configurations for a partial configuration (including uniform random sampling). Overall, we present 20 applications to motivate the usage of #SAT on feature models.

We presented algorithms and optimizations for each of the analyses listed above and uniform random sampling. The considered ideas consist of suggestions in the literature and new proposals that aim to reduce the number of required #SAT calls, accelerate #SAT calls, or both. We also propose algorithms that exploit the properties of a d-DNNF for each analysis.

For our empirical evaluation, we implemented and analyzed one algorithm for computing the number of valid configurations for a feature model, three algorithms for computing the commonalities of features, two algorithms for computing the number of remaining configurations of a partial configuration. For uniform random sampling, we implemented and analyzed three algorithms and additionally analyzed the tool KUS from Sharma et al. [SGRM18]. For the latter three analyses, we provided at least one algorithm that is based on exploiting the properties of a d-DNNF. The results show that the d-DNNF based computation were faster than the other algorithms for every analysis. Thus, we conclude that d-DNNF engines are a promising tool for counting based analyses of feature models. The results also indicated that computing core, dead, and false-optional features to reduce the number of required #SAT calls pays off. For every analysis, at least one algorithm and solver was able to evaluate the vast majority of models. Thus, we conclude that applications of #SAT solvers scale to industrial product lines. However, no solvers scales to two models including Linux.

We evaluated the analyses with twelve exact #SAT solvers. Five of those, namely `PicoSAT`, `Relsat`, `SharpCDCL`, `CNF2OBDD`, and `CNF2EADT`, could not even compute

https://doi.org/10.24355/dbbs.084-202009161329-0

the number of valid configurations for 10% of the feature models which we consider the threshold problem for the other analyses. Overall, `countAntom` requires the least time for the analyses that are not based on d-DNNF. However, there are analyses and feature models for which other solvers are faster. For the d-DNNF based analyses, we only considered solvers that produce smooth d-DNNFs, namely `c2d` and `dSharp`. `c2d` is not necessarily faster for creating a d-DNNF but creates smaller samples. Overall, the d-DNNF based algorithms are faster with `c2d`. This is a strong indicator that the additional effort to create smaller d-DNNFs pays off.

For some applications, estimating the number of valid configurations may be sufficient. Furthermore, the exact #SAT solvers do not scale to every feature model. Thus, we evaluated 2 approximate #SAT solvers, namely `ApproxMC` and `ApproxCount`, in addition to the 12 exact solvers. However, both solvers required significantly more time to analyze the feature models compared to the fastest exact solvers. We did not find benefits in estimating the number of valid configurations with both approximate #SAT solvers. However, it is possible that finding effective parameterizations for the solvers may reduce the required runtimes.

Overall, we conclude that (1) applying #SAT to feature models allows a variety of applications and, thus, is beneficial, (2) there is an algorithm for each #SAT dependent applications we considered that scales to a vast majority of industrial feature models, (3) a d-DNNF engine is a promising tool for counting based analyses, and (4) the performance of the algorithms is heavily dependent on the selection of the solver.

https://doi.org/10.24355/dbbs.084-202009161329-0

9. Future Work

In this chapter, we discuss open problems and potential future work for applying #SAT to feature models.

Further Optimize Exploitation of d-DNNFs

We presented algorithms that exploit the properties of a d-DNNF to compute analyses dependent on counting valid configurations in Chapter 4. These can be used to compute results for every application considered in this thesis. In addition, we show how to extract other information about the feature model from a d-DNNF (e.g., core/dead features and atomic sets). We argue that further research on d-DNNF for the analysis of feature models may be beneficial. First, the algorithms we present can still be optimized (e.g., skip unnecessary expensive arithmetic operations). Second, we expect that there are d-DNNF queries to allow even more analyses (e.g., detecting false-optional features). Third, the d-DNNF compilers translate the CNF without using any domain knowledge about the feature model. It may be beneficial to consider a translation that takes the feature into account.

Target Languages for Knowledge Compilation

Several target languages have been considered for the analysis of feature models. The most prevalent target language are binary decision diagrams [AHC+12, HPMFA+16, MWCC08, PLP11, MBC09, BSTC07, ACLF13]. Voronov et al. [VÅE11] exploit the properties of a decomposable negation normal form to enumerate valid configurations. Sharma et al. [SGRM18] use d-DNNFs for uniform random sampling. We also propose several algorithms dependent on d-DNNFs and showed that those perform well for the analysis of feature models. Darwiche et al. [DM02] discuss a variety of target languages with different properties. It is reasonable to assume that there are target languages that yield beneficial properties for the analysis of feature models that have not been considered yet. Identifying other promising target languages and examining their efficiency and capabilities on product lines may be beneficial.

https://doi.org/10.24355/dbbs.084-202009161329-0

Hard Feature Models

During our entire empirical evaluation, no #SAT solver, neither exact nor approximate, was able to evaluate the two feature models Linux and Automotive05. Thus, the applications considered in this thesis are not applicable to these models which motivates to identify ideas that are able to approximately evaluate hard models. We consider the following approaches to estimate a result for the models as future work: First, one may simplify the feature model by removing constraints until a solver is able to evaluate the adapted model. Second, we can compute the number of valid configurations induced by the feature model without cross-tree constraints which is possible with linear time complexity in the number of features [HGFACC11]. Then, domain knowledge (e.g., insights from earlier model versions) can be used for an educated guess on the limitations imposed by the cross-tree constraints to estimate a result. Third, the results can be estimated using approximate #SAT solvers parameterized specifically for the two models.

Parameterizations for Approximate #SAT Solvers

In our empirical evaluation, we considered two approximate #SAT solvers, namely `ApproxMC` and `ApproxCount`. Both solvers performed significantly worse than the best performing exact #SAT solvers. However, we only considered a single parameterization for the solvers for the entire evaluation. Adjusting parameters for approximate #SAT solvers may reduce their runtime and even enable the analysis of hard feature models, like Linux and Automotive05.

Required Accuracy of Approximated Results

The required accuracy of #SAT results differs for different applications. For example, detecting a faulty constraint that reduces the number of valid configurations by multiple orders of magnitude is easily possible with a deviation of 50% for the computed number of valid configurations. In contrast, for uniform random sampling, a deviation of 1% may cause not uniformly distributed or even faulty samples for the proposed algorithms. The examples show that the required accuracy vastly differs for different applications. To argue about the benefits of approximated results it is also important to examine the required accuracy for applications.

https://doi.org/10.24355/dbbs.084-202009161329-0

Bibliography

[ACLF13] Mathieu Acher, Philippe Collet, Philippe Lahire, and Robert B France. Familiar: A domain-specific language for large scale management of feature models. *Science of Computer Programming*, 78(6):657–681, 2013. (cited on Page 121)

[AGH05] Ken Arnold, James Gosling, and David Holmes. *The Java programming language*. Addison Wesley Professional, 2005. (cited on Page 57)

[AHC+12] Mathieu Acher, Patrick Heymans, Philippe Collet, Clément Quinton, Philippe Lahire, and Philippe Merle. Feature model differences. In *International Conference on Advanced Information Systems Engineering*, pages 629–645. Springer, 2012. (cited on Page 22, 117, and 121)

[AHKT+16] Mustafa Al-Hajjaji, Sebastian Krieter, Thomas Thüm, Malte Lochau, and Gunter Saake. Incling: efficient product-line testing using incremental pairwise sampling. *ACM SIGPLAN Notices*, 52(3):144–155, 2016. (cited on Page 73)

[AMS01] F Aloul, I Markov, and K Sakallah. Mince: A static global variable-ordering for sat and bdd. In *International Workshop on Logic and Synthesis*, pages 1167–1172, 2001. (cited on Page 38)

[AMS+18] Iago Abal, Jean Melo, Stefan Stănciulescu, Claus Brabrand, Márcio Ribeiro, and Andrzej Wąsowski. Variability Bugs in Highly Configurable Systems: A Qualitative Analysis. 26(3):10:1–10:34, January 2018. (cited on Page 53)

[Ana16] Sofia Ananieva. *Explaining Defects and Identifying Dependencies in Interrelated Feature Models*. PhD thesis, Institute of Software, 2016. (cited on Page 15 and 20)

[AT17] Dimitris Achlioptas and Panos Theodoropoulos. Probabilistic model counting with short xors. In *International Conference on Theory and Applications of Satisfiability Testing*, pages 3–19. Springer, 2017. (cited on Page 3)

[Bat05] Don Batory. Feature models, grammars, and propositional formulas. In *International Conference on Software Product Lines*, pages 7–20. Springer, 2005. (cited on Page 1, 17, and 135)

https://doi.org/10.24355/dbbs.084-202009161329-0

[BBMY04] Barry Boehm, A Winsor Brown, Ray Madachy, and Ye Yang. A software product line life cycle cost estimation model. In *Proceedings. 2004 International Symposium on Empirical Software Engineering, 2004. ISESE'04.*, pages 156–164. IEEE, 2004. (cited on Page 29)

[BDP03] Fahiem Bacchus, Shannon Dalmao, and Toniann Pitassi. Algorithms and complexity results for# sat and bayesian inference. In *44th Annual IEEE Symposium on Foundations of Computer Science, 2003. Proceedings.*, pages 340–351. IEEE, 2003. (cited on Page 11)

[BG19] Michele Boreale and Daniele Gorla. 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*, pages 363–378. Springer, 2019. (cited on Page 3, 13, and 95)

[BHvM09] Armin Biere, Marijn Heule, and Hans van Maaren. *Handbook of satisfiability*, volume 185. IOS press, 2009. (cited on Page 5, 11, 12, and 13)

[Bie08] Armin Biere. Picosat essentials. *Journal on Satisfiability, Boolean Modeling and Computation*, 4:75–97, 2008. (cited on Page 3, 11, 12, 36, 37, 92, and 136)

[BJP00] Roberto J Bayardo Jr and Joseph Daniel Pehoushek. Counting models using connected components. In *AAAI/IAAI*, pages 157–162, 2000. (cited on Page 1, 3, 11, 12, 36, 37, 92, 93, 112, 135, and 136)

[BSB15] Jan Burchard, Tobias Schubert, and Bernd Becker. Laissez-faire caching for parallel# sat solving. In *International Conference on Theory and Applications of Satisfiability Testing*, pages 46–61. Springer, 2015. (cited on Page 1, 3, 11, 12, 13, 37, 92, 93, 112, 135, and 136)

[BSRC10] David Benavides, Sergio Segura, and Antonio Ruiz-Cortés. Automated analysis of feature models 20 years later: A literature review. *Information Systems*, 35(6):615–636, 2010. (cited on Page 1, 2, 5, 13, 14, 15, 16, 17, 20, 23, 31, 117, and 135)

[BSTC07] David Benavides, Sergio Segura, Pablo Trinidad, and Antonio Ruiz Cortés. Fama: Tooling a framework for the automated analysis of feature models. *VaMoS*, 2007:01, 2007. (cited on Page 26, 117, and 121)

[BTRC05] David Benavides, Pablo Trinidad, and Antonio Ruiz-Cortés. Automated reasoning on feature models. In *International Conference on Advanced Information Systems Engineering*, pages 491–503. Springer, 2005. (cited on Page 1, 21, 117, and 135)

[BTS19] Paul Maximilian Bittner, Thomas Thüm, and Ina Schaefer. Sat encodings of the at-most-k constraint. In *International Conference on*

https://doi.org/10.24355/dbbs.084-202009161329-0

Software Engineering and Formal Methods, pages 127–144. Springer, 2019. (cited on Page 39)

[CE11] Sheng Chen and Martin Erwig. Optimizing the product derivation process. In *2011 15th International Software Product Line Conference*, pages 35–44. IEEE, 2011. (cited on Page 26 and 31)

[CFM+15] Supratik Chakraborty, Daniel J Fremont, Kuldeep S Meel, Sanjit A Seshia, and Moshe Y Vardi. On parallel scalable uniform sat witness generation. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 304–319. Springer, 2015. (cited on Page 31)

[Chv79] Vasek Chvatal. A greedy heuristic for the set-covering problem. *Mathematics of operations research*, 4(3):233–235, 1979. (cited on Page 73)

[CK05a] Donald Chai and Andreas Kuehlmann. A fast pseudo-boolean constraint solver. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 24(3):305–317, 2005. (cited on Page 51)

[CK05b] Krzysztof Czarnecki and Chang Hwan Peter Kim. Cardinality-based feature modeling and constraints: A progress report. In *International Workshop on Software Factories*, pages 16–20. ACM San Diego, California, USA, 2005. (cited on Page 23)

[CMC05] Paul C Clements, John D McGregor, and Sholom G Cohen. The structured intuitive model for product line economics (simple). Technical report, CARNEGIE-MELLON UNIV PITTSBURGH PA SOFTWARE ENGINEERING INST, 2005. (cited on Page 2, 24, 25, 29, and 30)

[CMV13] Supratik Chakraborty, Kuldeep S Meel, and Moshe Y Vardi. A scalable approximate model counter. In *International Conference on Principles and Practice of Constraint Programming*, pages 200–216. Springer, 2013. (cited on Page 95)

[CMV16] Supratik Chakraborty, Kuldeep S Meel, and Moshe Y Vardi. Algorithmic improvements in approximate counting for probabilistic inference: From linear to logarithmic sat calls. Technical report, 2016. (cited on Page 95)

[Coh03] Sholom Cohen. Predicting when product line investment pays. Technical report, CARNEGIE-MELLON UNIV PITTSBURGH PA SOFTWARE ENGINEERING INST, 2003. (cited on Page 29)

[CW07] Krzysztof Czarnecki and Andrzej Wasowski. Feature diagrams and logics: There and back again. In *11th International Software Product Line Conference (SPLC 2007)*, pages 23–34. IEEE, 2007. (cited on Page 1, 23, 117, and 135)

https://doi.org/10.24355/dbbs.084-202009161329-0

[Dar01a] Adnan Darwiche. Decomposable negation normal form. *Journal of the ACM (JACM)*, 48(4):608–647, 2001. (cited on Page 8 and 35)

[Dar01b] Adnan Darwiche. 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):11–34, 2001. (cited on Page 118)

[Dar02] Adnan Darwiche. A compiler for deterministic, decomposable negation normal form. In *AAAI/IAAI*, pages 627–634, 2002. (cited on Page 8, 9, 11, 36, 92, 93, 112, and 118)

[Dar04] Adnan Darwiche. New advances in compiling cnf to decomposable negation normal form. In *Proceedings of the 16th European Conference on Artificial Intelligence*, pages 318–322. Citeseer, 2004. (cited on Page 1, 2, 12, 37, 64, 92, 93, 135, and 136)

[DDM06] Bruno Dutertre and Leonardo De Moura. A fast linear-arithmetic solver for dpll (t). In *International Conference on Computer Aided Verification*, pages 81–94. Springer, 2006. (cited on Page 7)

[DK05] Vijay Durairaj and Priyank Kalla. Variable ordering for efficient sat search by analyzing constraint-variable dependencies. In *International Conference on Theory and Applications of Satisfiability Testing*, pages 415–422. Springer, 2005. (cited on Page 38)

[DM02] Adnan Darwiche and Pierre Marquis. A knowledge compilation map. *Journal of Artificial Intelligence Research*, 17:229–264, 2002. (cited on Page 5, 6, 7, 8, 9, 10, 11, 12, 35, and 121)

[DMB08] Leonardo De Moura and Nikolaj Bjørner. Z3: An efficient smt solver. In *International conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 337–340. Springer, 2008. (cited on Page 118)

[FAGS09] David Fernandez-Amoros, Ruben Heradio Gil, and Jose Cerrada Somolinos. Inferring information from feature diagrams to product line economic models. In *Proceedings of the 13th International Software Product Line Conference*, pages 41–50. Carnegie Mellon University, 2009. (cited on Page 2 and 115)

[FAHCC14] David Fernandez-Amoros, Ruben Heradio, Jose A Cerrada, and Carlos Cerrada. A scalable approach to exact model and commonality counting for extended feature models. *IEEE Transactions on Software Engineering*, 40(9):895–910, 2014. (cited on Page 21, 26, 29, 30, 43, 45, and 116)

[fea19] 2019. (cited on Page 3, 57, and 58)

[GAT+16] José A Galindo, Mathieu Acher, Juan Manuel Tirado, Cristian Vidal, Benoit Baudry, and David Benavides. Exploiting the enumeration of

https://doi.org/10.24355/dbbs.084-202009161329-0

all feature model configurations: A new perspective with distributed computing. In *Proceedings of the 20th International Systems and Software Product Line Conference*, pages 74–78. ACM, 2016. (cited on Page 1 and 135)

[GPFW96] Jun Gu, Paul W Purdom, John Franco, and Benjamin W Wah. Algorithms for the satisfiability (sat) problem: A survey. Technical report, Cincinnati Univ oh Dept of Electrical and Computer Engineering, 1996. (cited on Page 11)

[GSS06] Carla P Gomes, Ashish Sabharwal, and Bart Selman. Model counting: A new strategy for obtaining good bounds. In *AAAI*, pages 54–61, 2006. (cited on Page 3 and 11)

[HD03] Jinbo Huang and Adnan Darwiche. A structure-based variable ordering heuristic for sat. In *IJCAI*, volume 3, pages 1167–1172, 2003. (cited on Page 37 and 38)

[HD04] Jinbo Huang and Adnan Darwiche. Using dpll for efficient obdd construction. In *International Conference on Theory and Applications of Satisfiability Testing*, pages 157–172. Springer, 2004. (cited on Page 37)

[HFACA13] Ruben Heradio, David Fernandez-Amoros, Jose A Cerrada, and Ismael Abad. 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):1177–1204, 2013. (cited on Page 2, 11, 14, 17, 21, 24, 25, 28, 29, 115, and 117)

[HGFACC11] Rubén Heradio-Gil, David Fernandez-Amoros, José Antonio Cerrada, and Carlos Cerrada. Supporting commonality-based analysis of software product lines. *IET software*, 5(6):496–509, 2011. (cited on Page 2, 19, 20, 21, 28, 29, 116, and 122)

[HPMFA+16] Ruben Heradio, Hector Perez-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 *SoMeT*, pages 139–154, 2016. (cited on Page 42, 117, and 121)

[HR04] Michael Huth and Mark Ryan. *Logic in Computer Science: Modelling and reasoning about systems*. Cambridge university press, 2004. (cited on Page 6, 9, 10, and 11)

[HS00] Holger H Hoos and Thomas Stützle. Local search algorithms for sat: An empirical evaluation. *Journal of Automated Reasoning*, 24(4):421–481, 2000. (cited on Page 10)

https://doi.org/10.24355/dbbs.084-202009161329-0

[HSJ+04] Tarik Hadzic, Sathiamoorthy Subbarayan, Rune M Jensen, Henrik R Andersen, Jesper Møller, and Henrik Hulgaard. Fast backtrack-free product configuration using a precompiled solution space representation. *small*, 10(1):3, 2004. (cited on Page 9)

[Joh92] David S Johnson. The np-completeness column: an ongoing guide. *Journal of algorithms*, 13(3):502–524, 1992. (cited on Page 11 and 13)

[JRL08] Narendra Jussien, Guillaume Rochart, and Xavier Lorca. Choco: an open source java constraint programming library. 2008. (cited on Page 118)

[JSRM+18] Paulius Juodisius, Atrisha Sarkar, Raghava Rao Mukkamala, Michal Antkiewicz, Krzysztof Czarnecki, and Andrzej Wasowski. Clafer: Lightweight modeling of structure, behaviour, and variability. *arXiv preprint arXiv:1807.08576*, 2018. (cited on Page 118)

[KK07] Will Klieber and Gihwon Kwon. Efficient cnf encoding for selecting 1 from n objects. In *Proc. International Workshop on Constraints in Formal Verification*, 2007. (cited on Page 39)

[KLMT13] Frédéric Koriche, Jean-Marie Lagniez, Pierre Marquis, and Samuel Thomas. Knowledge compilation for model counting: Affine decision trees. In *Twenty-Third International Joint Conference on Artificial Intelligence*, 2013. (cited on Page 3, 92, 94, and 136)

[KMM13] Vladimir Klebanov, Norbert Manthey, and Christian Muise. Sat-based analysis and quantification of information flow in programs. In *International Conference on Quantitative Evaluation of Systems*, pages 177–192. Springer, 2013. (cited on Page 3, 11, 92, 93, and 136)

[KTM+17] 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 *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*, pages 291–302. ACM, 2017. (cited on Page 20, 91, 92, and 111)

[KTS+20] Sebastian Krieter, Thomas Thüm, Sandro Schulze, Gunter Saake, and Thomas Leich. Yasa: yet another sampling algorithm. In *Proceedings of the 14th International Working Conference on Variability Modelling of Software-Intensive Systems*, pages 1–10, 2020. (cited on Page 32)

[KZK10] Andreas Kübler, Christoph Zengler, and Wolfgang Küchlin. Model counting in product configuration. *arXiv preprint arXiv:1007.1024*, 2010. (cited on Page 1, 2, 5, 11, 13, 16, 19, 22, 26, 29, 41, 115, 116, 117, and 135)

[LGCR15] Jia Hui Liang, Vijay Ganesh, Krzysztof Czarnecki, and Venkatesh Raman. Sat-based analysis of large real-world feature models is easy. In

https://doi.org/10.24355/dbbs.084-202009161329-0

Proceedings of the 19th International Conference on Software Product Line, pages 91–100, 2015. (cited on Page 2)

[Lib00] Paolo Liberatore. On the complexity of choosing the branching literal in dpll. *Artificial intelligence*, 116(1-2):315–326, 2000. (cited on Page 6 and 10)

[LM17] Jean-Marie Lagniez and Pierre Marquis. An improved decision-dnnf compiler. In *IJCAI*, pages 667–673, 2017. (cited on Page 1, 3, 7, 8, 12, 36, 37, 64, 92, 94, 112, 118, 135, and 136)

[MBC09] Marcilio Mendonca, Moises Branco, and Donald Cowan. Splot: software product lines online tools. In *Proceedings of the 24th ACM SIGPLAN conference companion on Object oriented programming systems languages and applications*, pages 761–762. ACM, 2009. (cited on Page 121)

[MDSD14] Raúl Mazo, Cosmin Dumitrescu, Camille Salinesi, and Daniel Diaz. Recommendation heuristics for improving product line configuration processes. In *Recommendation Systems in Software Engineering*, pages 511–537. Springer, 2014. (cited on Page 30 and 31)

[MFM04] Yogesh S Mahajan, Zhaohui Fu, and Sharad Malik. Zchaff2004: An efficient sat solver. In *International Conference on Theory and Applications of Satisfiability Testing*, pages 360–375. Springer, 2004. (cited on Page 2)

[MKR+16] Flávio Medeiros, Christian Kästner, Márcio Ribeiro, Rohit Gheyi, and Sven Apel. A comparison of 10 sampling algorithms for configurable systems. In *2016 IEEE/ACM 38th International Conference on Software Engineering (ICSE)*, pages 643–654. IEEE, 2016. (cited on Page 31)

[MMBH10] Christian Muise, Sheila McIlraith, J Christopher Beck, and Eric Hsu. Fast d-dnnf compilation with sharpsat. In *Workshops at the twenty-fourth AAAI conference on artificial intelligence*, 2010. (cited on Page 3, 5, 9, 12, 36, 37, 64, 92, 94, 112, 118, and 136)

[MMZ+01] Matthew W Moskewicz, Conor F Madigan, Ying Zhao, Lintao Zhang, and Sharad Malik. Chaff: Engineering an efficient sat solver. In *Proceedings of the 38th annual Design Automation Conference*, pages 530–535, 2001. (cited on Page 37)

[MOP+19] 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 *Proceedings of the 23rd International Systems and Software Product Line Conference-Volume A*, page 39. ACM, 2019. (cited on Page 1, 7, 19, 32, 117, 118, 135, and 136)

https://doi.org/10.24355/dbbs.084-202009161329-0

[MTS+17] Jens Meinicke, Thomas Thüm, Reimar Schröter, Fabian Benduhn, Thomas Leich, and Gunter Saake. *Mastering software variability with FeatureIDE*. Springer, 2017. (cited on Page 92 and 112)

[MWC09] Marcilio Mendonca, Andrzej Wąsowski, and Krzysztof Czarnecki. Sat-based analysis of feature models is easy. In *Proceedings of the 13th International Software Product Line Conference*, pages 231–240. Carnegie Mellon University, 2009. (cited on Page 1, 17, and 135)

[MWCC08] Marcilio Mendonca, Andrzej Wasowski, Krzysztof Czarnecki, and Donald Cowan. Efficient compilation techniques for large scale feature models. In *Proceedings of the 7th international conference on Generative programming and component engineering*, pages 13–22. ACM, 2008. (cited on Page 1, 38, 117, 121, and 135)

[NdAM08] Jarley Palmeira Nóbrega, Eduardo Santana de Almeida, and Sílvio Romero Lemos Meira. Income: Integrated cost model for product line engineering. In *2008 34th Euromicro Conference Software Engineering and Advanced Applications*, pages 27–34. IEEE, 2008. (cited on Page 29)

[NOT06] Robert Nieuwenhuis, Albert Oliveras, and Cesare Tinelli. Solving sat and sat modulo theories: From an abstract davis–putnam–logemann–loveland procedure to dpll (t). *Journal of the ACM (JACM)*, 53(6):937–977, 2006. (cited on Page 5)

[NW01] Andreas Nonnengart and Christoph Weidenbach. Computing small clause normal forms. *Handbook of automated reasoning*, 1(335-367):3, 2001. (cited on Page 37)

[OBMS16] Jeho Oh, Don Batory, Margaret Myers, and Norbert Siegmund. Finding product line configurations with high performance by random sampling. Technical report, Technical Report TR-16-22. University of Texas at Austin, Department of . . . , 2016. (cited on Page 19, 32, 49, 51, 54, and 55)

[OBMS17] Jeho Oh, Don Batory, Margaret Myers, and Norbert Siegmund. Finding near-optimal configurations in product lines by random sampling. In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*, pages 61–71. ACM, 2017. (cited on Page 1, 19, 32, 49, 51, 116, and 117)

[OD15] Umut Oztok and Adnan Darwiche. A top-down compiler for sentential decision diagrams. In *Twenty-Fourth International Joint Conference on Artificial Intelligence*, 2015. (cited on Page 3, 92, 94, and 136)

[OGB+19] Jeho Oh, Paul Gazzillo, Don Batory, Marijn Heule, and Maggie Myers. Uniform sampling from kconfig feature models. *The University of Texas at Austin, Department of Computer Science, Tech. Rep. TR-19-02*, 2019. (cited on Page 1, 7, 13, 19, 31, 35, 37, 38, 51, 52, 53, 91, 112, 116, 117, 135, and 136)

https://doi.org/10.24355/dbbs.084-202009161329-0

[PG86] David A Plaisted and Steven Greenbaum. A structure-preserving clause form translation. *Journal of Symbolic Computation*, 2(3):293–304, 1986. (cited on Page 13)

[PHRC06] Joaquin Pena, 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. 2006. (cited on Page 26)

[PLP11] Richard Pohl, Kim Lauenroth, and Klaus Pohl. 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*, pages 313–322. IEEE Computer Society, 2011. (cited on Page 1, 2, 13, 14, 19, 117, 121, and 135)

[PM16] Héctor José Pérez Morago. Bdd algorithms to perform hard analysis operations on variability models. 2016. (cited on Page 26)

[PSK+10] Gilles Perrouin, Sagar Sen, Jacques Klein, Benoit Baudry, and Yves Le Traon. Automated and scalable t-wise test case generation strategies for software product lines. In *2010 Third international conference on software testing, verification and validation*, pages 459–468. IEEE, 2010. (cited on Page 1 and 135)

[SAKS16] Stefan Sobernig, Sven Apel, Sergiy Kolesnikov, and Norbert Siegmund. Quantifying structural attributes of system decompositions in 28 feature-oriented software product lines. *Empirical Software Engineering*, 21(4):1670–1705, 2016. (cited on Page 1, 13, and 135)

[SBB+04] Tian Sang, Fahiem Bacchus, Paul Beame, Henry A Kautz, and Toniann Pitassi. Combining component caching and clause learning for effective model counting. *SAT*, 4:7th, 2004. (cited on Page 36, 92, 93, and 112)

[SBK05a] Tian Sang, Paul Beame, and Henry Kautz. Heuristics for fast exact model counting. In *International Conference on Theory and Applications of Satisfiability Testing*, pages 226–240. Springer, 2005. (cited on Page 3, 11, 12, 37, 92, 93, 112, and 136)

[SBK05b] Tian Sang, Paul Beame, and Henry Kautz. Solving bayesian networks by weighted model counting. In *Proceedings of the Twentieth National Conference on Artificial Intelligence (AAAI-05)*, volume 1, pages 475–482. AAAI Press, 2005. (cited on Page 43)

[Seg08] Sergio Segura. Automated analysis of feature models using atomic sets. In *SPLC (2)*, pages 201–207, 2008. (cited on Page 1, 17, and 135)

[SGRM18] Shubham Sharma, Rahul Gupta, Subhajit Roy, and Kuldeep S Meel. Knowledge compilation meets uniform sampling. In *LPAR*, pages 620–636, 2018. (cited on Page 53, 54, 95, 109, 117, 119, and 121)

https://doi.org/10.24355/dbbs.084-202009161329-0

[SKT+16] Reimar Schröter, Sebastian Krieter, Thomas Thüm, Fabian Benduhn, and Gunter Saake. Feature-model interfaces: the highway to compositional analyses of highly-configurable systems. In *Proceedings of the 38th International Conference on Software Engineering*, pages 667–678. ACM, 2016. (cited on Page 1, 42, and 135)

[SS03] João P Marques Silva and Karem A Sakallah. Grasp—a new search algorithm for satisfiability. In *The Best of ICCAD*, pages 73–89. Springer, 2003. (cited on Page 38)

[SSK+20] Joshua Sprey, Chico Sundermann, Sebastian Krieter, Michael Nieke, Jacopo Mauro, Thomas Thüm, and Ina Schaefer. Smt-based variability analyses in featureide. In *Proceedings of the 14th International Working Conference on Variability Modelling of Software-Intensive Systems*, pages 1–9, 2020. (cited on Page 1, 17, 30, 31, and 117)

[STS20] Chico Sundermann, Thomas Thüm, and Ina Schaefer. Evaluating# sat solvers on industrial feature models. In *Proceedings of the 14th International Working Conference on Variability Modelling of Software-Intensive Systems*, pages 1–9, 2020. (cited on Page 1, 3, 19, 22, 24, 29, 35, 39, 49, 68, 92, 95, 108, 111, and 117)

[TBC06] Pablo Trinidad, David Benavides, and Antonio Ruiz Cortés. Isolated features detection in feature models. In *CAiSE Forum*, 2006. (cited on Page 26)

[TBK09] Thomas Thum, Don Batory, and Christian Kastner. Reasoning about edits to feature models. In *2009 IEEE 31st International Conference on Software Engineering*, pages 254–264. IEEE, 2009. (cited on Page 22 and 40)

[TBW04] Christian Thiffault, Fahiem Bacchus, and Toby Walsh. Solving non-clausal formulas with dpll search. In *International Conference on Principles and Practice of Constraint Programming*, pages 663–678. Springer, 2004. (cited on Page 6)

[Thu06] Marc Thurley. sharpsat–counting models with advanced component caching and implicit bcp. In *International Conference on Theory and Applications of Satisfiability Testing*, pages 424–429. Springer, 2006. (cited on Page 1, 3, 11, 12, 36, 37, 92, 93, 112, 118, 135, and 136)

[Tiu98] Alwen Tiu. Introduction to logic. 1998. (cited on Page 36)

[TKB+14] Thomas Thüm, Christian Kästner, Fabian Benduhn, Jens Meinicke, Gunter Saake, and Thomas Leich. Featureide: An extensible framework for feature-oriented software development. *Science of Computer Programming*, 79:70–85, 2014. (cited on Page 73)

[TKES11] Thomas Thum, Christian Kastner, Sebastian Erdweg, and Norbert Siegmund. Abstract features in feature modeling. In *2011 15th International Software Product Line Conference*, pages 191–200. IEEE, 2011. (cited on Page 25)

https://doi.org/10.24355/dbbs.084-202009161329-0

[TS16] Takahisa Toda and Takehide Soh. Implementing efficient all solutions sat solvers. *Journal of Experimental Algorithmics (JEA)*, 21:1–12, 2016. (cited on Page 1, 3, 11, 92, 94, and 136)

[Tse83] Grigori S Tseitin. On the complexity of derivation in propositional calculus. In *Automation of reasoning*, pages 466–483. Springer, 1983. (cited on Page 7, 13, and 37)

[VÅE11] Alexey Voronov, Knut Åkesson, and Fredrik Ekstedt. Enumeration of valid partial configurations. In *Proceedings of Workshop on Configuration, IJCAI 2011*, volume 755, pages 25–31, 2011. (cited on Page 117 and 121)

[VAHT+18] Mahsa Varshosaz, Mustafa Al-Hajjaji, Thomas Thüm, Tobias Runge, Mohammad Reza Mousavi, and Ina Schaefer. A Classification of Product Sampling for Software Product Lines. In *Proceedings of the International Software Product Line Conference (SPLC)*, pages 1–13, September 2018. (cited on Page 53)

[VD10] Bart Veer and John Dallaway. The ecos component writer's guide. *seen Mar*, 2010. (cited on Page 92)

[WBS+10] Jules White, David Benavides, Douglas C Schmidt, Pablo Trinidad, Brian Dougherty, and Antonio Ruiz-Cortes. Automated diagnosis of feature model configurations. *Journal of Systems and Software*, 83(7):1094–1107, 2010. (cited on Page 1 and 135)

[Wel82] William J Welch. Algorithmic complexity: three np-hard problems in computational statistics. *Journal of Statistical Computation and Simulation*, 15(1):17–25, 1982. (cited on Page 11)

[WJHS04] Chao Wang, HoonSang Jin, Gary D Hachtel, and Fabio Somenzi. Refining the sat decision ordering for bounded model checking. In *Proceedings of the 41st annual Design Automation Conference*, pages 535–538, 2004. (cited on Page 38)

[WS05] Wei Wei and Bart Selman. A new approach to model counting. In *International Conference on Theory and Applications of Satisfiability Testing*, pages 324–339. Springer, 2005. (cited on Page 95)

[ZZM04] Wei Zhang, Haiyan Zhao, and Hong Mei. A propositional logic-based method for verification of feature models. In *International Conference on Formal Engineering Methods*, pages 115–130. Springer, 2004. (cited on Page 17)

https://doi.org/10.24355/dbbs.084-202009161329-0

https://doi.org/10.24355/dbbs.084-202009161329-0

Topic Description

Introduction

Product lines represent a family of similar products [BSRC10]. Hereby, the products are decomposed in smaller subsets, also called features [SAKS16]. A product can then be derived by a selection of features. Feature models are commonly used to specify all possible products of a product line. Such models consist of hierarchical structure of features and propositional cross-tree constraints [BSRC10, CW07, Bat05].

In general, analyzing a feature model is infeasible manually, as it is difficult to keep track of all dependencies between features. Therefore, automated support is required [Bat05]. Analyses considered in the literature typically rely on satisfaction based solvers like SAT solvers [Bat05, SKT+16, MWC09, PLP11, CW07, PSK+10, Seg08, GAT+16], CSP solvers [BTRC05, PLP11, Seg08, WBS+10], and binary decision diagrams [MWCC08, PLP11, Seg08, GAT+16, CW07]. Another possible type of solvers, that can be considered for feature model analysis are #SAT solvers which made vast advances in the last decade [BJP00, Thu06, BSB15, Dar04, LM17]. A #SAT solver computes the number of valid solutions for a given propositional formula, while a satisfaction based solver computes whether there is at least one solution.

A feature model can be translated to an equivalent propositional formula [MWC09]. Using such a formula as input for a #SAT solver computes the number of products of the underlying product line. Older #SAT solvers were only capable of analyzing smaller feature models [PLP11, KZK10]. Due to the recent advances, #SAT might be suitable to analyze feature models and even enable new analyses. The literature already considers some methods dependent on computing the number of products (e.g., variability reduction [BTRC05] and uniform random sampling [OGB+19, MOP+19]).

While it is known that SAT solvers scale for various feature model analyses [MWC09], research is still required regarding #SAT solvers. The goal of this master's thesis is to identify new applications for #SAT solvers and the examine the scalability. Therefore, we implement and evaluate the runtime and memory usage of the found applications with different #SAT solvers and input models.

Furthermore, we aim to improve the scalability of the applications using multiple optimizations. First, we aim to switch solvers depending on the algorithm and input. Second, our goal is to optimize the algorithms (e.g., by reducing the number

https://doi.org/10.24355/dbbs.084-202009161329-0

of required #SAT calls). Current #SAT solvers are not optimized for feature models. Optimizations that specifically exploit properties of the feature model might improve the scalability of a #SAT solver. Furthermore, none of the current #SAT solvers supports incremental queries [Dar04, LM17, MMBH10, BJP00, SBK05a, Thu06, BSB15, Bie08, KLMT13, KMM13, TS16, OD15]. However, some applications, like uniform random sampling [OGB+19, MOP+19], require a high number of similar solver calls. An algorithm that computes such samples might vastly benefit from an incremental solver. Such optimizations could be realized by a new solver or optimized input encodings.

Tasks

Mandatory

- **Gathering Applications:** We aim to examine possible applications of #SAT solvers for the analysis of feature models. In the first stage, we study applications considered in the literature. In the second stage, we try to work out new-found applications.
- **Concept:** For each new found relevant application, we develop at least one algorithm to compute the desired result for the application.
- **Implementation based on FeatureIDE:** We integrate the relevant developed algorithms corresponding to an application of #SAT solvers into the feature modelling tool FeatureIDE.
- **Evaluation:** We evaluate the applications using multiple #SAT solvers regarding the runtime and memory usage. For the evaluation, we consider automatically generated and real-world feature models.

Optional

- **Meta-Solver** We aim to create a meta-solver that switches solvers depending on the algorithm and the input.
- **Optimize Algorithms** We aim to optimize the algorithms regarding runtime and memory usage.
- **Optimize Input Encodings:** We aim to improve the scalability of solvers might be improved without directly changing the solver.
- **Create own Optimized #SAT Solver:** We aim to create a #SAT solver that is specifically optimized for feature models. Furthermore, we aim to support incremental queries.

Supervision

The thesis is supervised by Dr.-Ing. Thomas Thüm, Michael Nieke, and Prof. Dr.-Ing. Ina Schaefer, Institute of Software Engineering and Automotive Informatics.

https://doi.org/10.24355/dbbs.084-202009161329-0

Hiermit erkläre ich, dass ich die vorliegende Arbeit selbständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel verwendet habe.

Braunschweig, den 22. Juli 2020

https://doi.org/10.24355/dbbs.084-202009161329-0