Empirical Software Engineering (2023) 28:29
https://doi.org/10.1007/s10664-022-10265-9

Evaluating state-of-the-art #SAT solvers on industrial configuration spaces

Chico Sundermann[1] · Tobias Heß[1] · Michael Nieke[2] · Paul Maximilian Bittner[1] · Jeffrey M. Young[3] · Thomas Thüm[1] · Ina Schaefer[2]

Accepted: 25 November 2022 / Published online: 13 January 2023

Abstract
Product lines are widely used to manage families of products that share a common base of features. Typically, not every combination (configuration) of features is valid. Feature models are a de facto standard to specify valid configurations and allow standardized analyses on the variability of the underlying system. A large variety of such analyses depends on computing the number of valid configurations. To analyze feature models, they are typically translated to propositional logic. This allows to employ #SAT solvers that compute the number of satisfying assignments of the propositional formula translated from a feature model. However, the #SAT problem is generally assumed to be even harder than SAT and its scalability when applied to feature models has only been explored sparsely. Our main contribution is an investigation of the performance of off-the-shelf #SAT solvers on computing the number of valid configurations for industrial feature models. We empirically evaluate 21 publicly available #SAT solvers on 130 feature models from 15 subject systems. Our results indicate that current solvers master a majority of the evaluated systems (13/15) with the fastest solvers requiring less than one second for each successfully evaluated feature model. However, there are two complex systems for which none of the evaluated solvers scales. For the given experiment design, the solvers that consumed the least runtime are `sharpSAT` (2.5 seconds in sum for the 13 systems) and `Ganak` (3.5 seconds).

Keywords Configurable systems · Feature models · Product lines · Model counting · Configuration counting · #SAT · Benchmark

1 Introduction

A product line represents a family of products that share certain configuration options, also called features (Benavides et al. 2010; Sobernig et al. 2016; Bosch et al. 2001; Heradio et al.

Communicated by: Philippe Collet, Sarah Nadi, Christoph Seidl

This article belongs to the Topical Collection: *Special Issue on Software Product Lines and Variability-rich Systems (SPLC).*

✉ Chico Sundermann
chico.sundermann@uni-ulm.de

Extended author information available on the last page of the article.

2019). Each product is composed of a distinct selection of features, called configuration (Apel et al. 2013). However, systems typically contain constraints which limit the set of valid configurations (e.g., the selection of one feature requires selecting another feature). These constraints are typically specified as a feature model (Bagheri et al. 2012; Batory 2005; Czarnecki and Wa̧sowski 2007) which consists of a tree hierarchy and additional cross-tree constraints.

Managing a product line is typically complex due to the high number of constraints (Benavides et al. 2010). For example, one feature model we analyzed, representing an automotive product line, contains more than 10,000 cross-tree constraints in addition to hierarchical constraints. Manually keeping track of all these dependencies is infeasible (Sprey et al. 2020). Consequently, a large variety of automated support in terms of analyses has been proposed (Batory 2005; Schröter et al. 2016; Mendonça et al. 2009; Pohl et al. 2011; Czarnecki and Wa̧sowski 2007; Perrouin et al. 2010; Segura 2008; Galindo et al. 2016; Benavides et al. 2010; Sprey et al. 2020). A multitude of analyses is based on feature-model counting (i.e., computing the number of valid configurations), such as uniform random sampling (Munoz et al. 2019; Oh et al. 2019; Sharma et al. 2018) and detecting design errors (Sundermann et al. 2021; Heradio et al. 2013; Chen and Erwig 2011; Fernández-Amorós et al. 2014; Heradio-Gil et al. 2011; Kübler et al. 2010). We refer to the number of valid configurations of a feature model as its *cardinality* (Sundermann et al. 2021).

In the literature, the scalability of analyses that depend on computing the number of valid configurations is largely unknown. Existing work either focuses on single analyses (e.g., uniform random sampling of feature-model configurations (Oh et al. 2017, 2019; Sharma et al. 2018)), has not been evaluated on industrial feature models (Heradio-Gil et al. 2011; Fernández-Amorós et al. 2014; Pohl et al. 2011), or considers very few solvers or systems (Kübler et al. 2010; Oh et al. 2017, 2019; Sharma et al. 2018). In this paper, we focus on propositional model counting (for short #SAT) which determines the number of satisfying assignments for a given propositional formula. As the translation of feature models to propositional logic is well-researched (Benavides et al. 2010; Batory 2005), #SAT solvers can be applied out of the box to compute the cardinality of feature models. However, #SAT is at least as hard as SAT because after computing #SAT (i.e., the number of satisfying assignments) it is trivial to determine whether a formula is SAT (i.e., there is at least one satisfying assignment). In general, #SAT is assumed to be harder (Burchard et al. 2015; Valiant 1979). While it is widely accepted that regular SAT is typically easy for industrial feature models (compared to randomly generated formulas (Pett et al. 2019; Mendonça et al. 2009)), this has not been explored for #SAT.

In this work, we provide insights on the scalability of modern off-the-shelf #SAT solvers for the analysis of feature models. Analyses based on feature-model counting can only be applied in practice if available #SAT solvers scale to industrial feature models considering time restrictions for typical use cases, such as interactive settings (Fritsch et al. 2020; Sprey et al. 2020; Krieter et al. 2017; Acher et al. 2013; Benavides et al. 2007) or continuous integration environments (Pett et al. 2021). We thus evaluate the runtimes of analyzing feature models with publicly available #SAT solvers. Furthermore, we provide recommendations on which solvers to use for analyzing feature models to reduce runtimes.

#SAT solvers rely on a variety of techniques to compute the number of satisfying assignments. While some solvers only report the number of satisfying assignments (Bayardo Jr and Pehoushek 2000; Sang et al. 2004; Thurley 2006; Burchard et al. 2015; Biere 2008), other solvers apply knowledge compilation to different target languages, such as *binary decision diagrams* (BDDs) http://buddy.sourceforge.net/manual/main.html. Accessed: 02

Mar 2020; https://github.com/vscosta/cudd. Accessed: 13 Jun 2020 (Toda and Soh 2016), *deterministic decomposable negation normal forms* (d-DNNFs) (Darwiche 2004; Lagniez and Marquis 2017; Muise et al. 2010), *sentential decision diagrams* (SDDs) (Oztok and Darwiche 2015), and *extended affine decision trees* (Koriche et al. 2013). The compiled target languages may be reused for further feature-model analyses. We analyze the benefits of different techniques to identify promising classes of #SAT solvers.

In general, the runtime required to analyze a feature model depends on structural properties related to its size and complexity (Mendonça et al. 2009; Kübler et al. 2010; Fernández-Amorós et al. 2014). We provide first insights on properties which induce a time-consuming computation for every or some #SAT solvers. In particular, we analyze the correlation between the runtimes and a variety of structural metrics.

For some feature models, it may be infeasible to compute an exact result using publicly available solvers. In this case, approximate #SAT solvers, which estimate the number of satisfying assignments for a given formula, may be beneficial. We inspect the benefits of approximate #SAT solvers when applied to industrial feature models.

Overall, we evaluate 19 exact and 2 approximate off-the-shelf #SAT solvers which are publicly available. For our empirical evaluation, we consider 15 subject systems with overall 130 feature models. We provide the framework and data used for the empirical evaluation on Zenodo.[1] In particular, our work provides the following contributions:

1. We examine the performance regarding runtime of #SAT technology on 130 industrial feature models.
2. We identify best performing #SAT solvers out of 21 off-the-shelf tools.
3. We compare the benefits of different #SAT technologies.
4. We examine the correlation between the runtime of #SAT solvers and structural metrics of the feature model.
5. We inspect the performance of two approximate #SAT solvers.
6. We provide the number of valid configurations for feature models in our dataset.

In this work, we extend our previous conference publication (Sundermann et al. 2020) regarding the following aspects. First, we additionally evaluate ten more exact #SAT solvers. Second, we examine the runtimes of two approximate #SAT solvers. Third, we consider four additional subject systems. Fourth, we analyze the correlation between the runtime and 12 structural metrics of the feature models. Fifth, we improve the accuracy of our results by repeating the measurements and applying statistical tests to study the significance of our results. Overall, the evaluation subsumes the previous evaluation (Sundermann et al. 2020) except for analyzing the evolution of systems. We consider a more thorough analysis (compared to the previous evaluation (Sundermann et al. 2020)) of the evolution as out of scope for this work.

2 Motivating example

Figure 1 shows a feature diagram representing a simplified car product line. A feature diagram is a commonly used visual representation of a feature model (Benavides et al. 2010). It visualizes the feature model's tree structure and additional cross-tree constraints given in

[1] https://doi.org/10.5281/zenodo.7329979

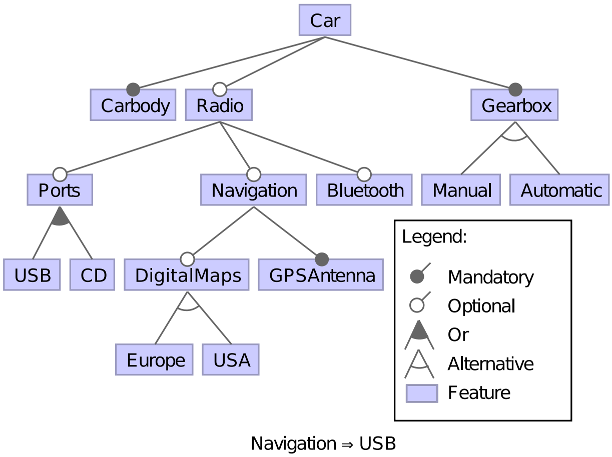

Fig. 1 Example feature model adapted from Ananieva et al. (Ananieva et al. 2016)

propositional logic. The tree structure and the cross-tree constraints specify the set of valid configurations.

In our example, each car of the product line requires a *Carbody*. This is indicated by the mandatory property of the feature. In contrast, a *Radio* is an optional feature (i.e., it may or may not be selected). A configuration that does not contain exactly one of the *Gearbox* types, *Manual* or *Automatic*, is invalid, as they appear in an alternative-relation in the feature diagram. Furthermore, the *Ports* of a *Radio* include at least one of *USB* or *CD*. This relation is described by an or-relation. The cross-tree constraint *Navigation* ⇒ *USB* represents that a car with *Navigation* requires a *USB* port.

To analyze a feature model, we can use its cardinality (i.e., the number of valid configurations). Consider the following scenario. The vast majority of automatic cars are sold in the USA. As a consequence, a developer introduces a new constraint $Automatic \Rightarrow USA$ (automatic cars require digital maps for the USA). Using a #SAT solver, the developer finds that the cardinality is 42 before the change and 25 afterwards. The immense decrease in the cardinality, if unexpected, may already be an indicator for a design problem, because the set of available cars is almost halved. Further, the cardinality can be combined with domain knowledge for more sophisticated insights as follows. In the old version, there are 21 cars with an *Automatic* gearbox and 21 cars with a *Manual* gearbox. The newly introduced constraint has no impact on cars with *Manual*. Thus, there are still 21 cars with *Manual* in the new version which implies that only $25 - 21 = 4$ valid configurations with *Automatic* remain. Due to the tree hierarchy, the introduced constraint ($Automatic \Rightarrow USA$) requires each automatic car to also have *Radio*, *Navigation*, *DigitalMaps*, and *USB*. This side effect was probably unintended and can be fixed by changing the constraint to $Automatic \wedge DigitalMaps \Rightarrow USA$. While the original constraint ($Automatic \Rightarrow USA$) induces an immense and possibly unintended reduction in the variability, it introduces no traditional anomalies (e.g., core, false-optional, or dead features; cf. (Benavides et al. 2010) for more details). Hence, in such cases it is hard to detect the side effects with traditional SAT-based analyses. In the provided scenario, we used the variability reduction of a feature model update to detect side effects, one of 21 applications of #SAT we identified in previous work (Sundermann et al. 2021).

Without cross-tree constraints, computing the number of valid configurations has linear time complexity in the number of features (Heradio-Gil et al. 2011). Only considering the tree-structure, the selections in a subtree are completely independent of selections in other subtrees. Therefore, the cardinality of each subtree can be computed separately. The cardinality of the feature model can be computed by traversing the tree once and applying rules for each relation type, recursively. For example, the cardinality of an alternative group is equal to the sum of cardinalities of the subtrees induced by the children of the alternative group. However, for feature models with cross-tree constraints, the proposed procedure results in a wrong cardinality as interdependencies are disregarded. Hence, a more sophisticated algorithm is required.

With cross-tree constraints, the number of configurations cannot be computed in linear time complexity w.r.t. to the number of features. Every feature model can be translated to a propositional formula (Mendonça et al. 2009). Furthermore, a feature model that contains cross-tree constraints can represent every propositional formula and vice versa (Knüppel et al. 2017). Thus, computing the satisfiability of a model with those constraints is as hard as SAT and computing the number of valid configurations is as hard as #SAT.

3 The need for feature-model counting

In our previous work (Sundermann et al. 2021), we surveyed a large variety of applications dependent on the number of valid configurations of feature models. The presented applications indicate the benefits of applying #SAT to feature models for multiple aspects, such as detecting design errors, economical estimations, and guidance for developers. Overall, we found 21 applications gathered from the literature or inspired by industry projects, one of which we exemplified in the last section. In the following, we present some exemplary applications that depend on computing the number of valid configurations provided in the original work (Sundermann et al. 2021). Each of the exemplary applications is inspired by insights of our industry projects.

Variability reduction In Section 2, we already introduced an example of variability reduction to detect the side effect of a new constraint. Generally, when working with product lines, it is infeasible to manually keep track of all possible side effects when applying changes (Sprey et al. 2020; Chen and Erwig 2011; Heradio et al. 2016). These side effects are especially difficult to detect if they introduce no traditional anomaly, such as dead features (Benavides et al. 2010), or a void feature model (Benavides et al. 2010) (i.e., the feature model does not describe a single valid configuration). In such cases, computing the cardinality before and after a change may provide an indicator for faulty edits (Sundermann et al. 2021). Another use case is willingly decreasing the cardinality to limit the variability of a system during an evolution. However, in order to grasp the impact of such changes it is necessary to know the cardinality before and after the change (Benavides et al. 2005).

Feature prioritization In some scenarios, features can be prioritized based on the number of valid configurations they appear in. For example, a developer may have to decide which feature to develop next. Suppose the developer's goal is to develop as many distinct products as possible. Consequently, the developer wants to prioritize features that appear in a higher number of valid configurations, which can be computed using a #SAT solver (Sundermann et al. 2021).

Uniform random sampling As it is mostly infeasible to analyze a configuration space by considering each single configuration, it is common to create representative samples for a product line (Munoz et al. 2019). However, finding these samples is not trivial (Oh et al. 2016, 2017). Uniform random sampling creates representative (i.e., each valid configuration has the same chance to be included) samples (Munoz et al. 2019). One technique for uniform random sampling is to create a bijection between integers and valid configurations. Suppose the cardinality of the feature model is #FM. Then, by randomly selecting an integer within the range [1,. . .,#FM], each configuration has the same probability to be included in the sample. The bijection can be achieved using #SAT by recursively assigning the features (Munoz et al. 2019). Following the algorithm of Munoz et al. (Oh et al. 2019), the number of valid configurations needs to be computed for each assignment in the worst case. This requires an efficient #SAT solver, especially for large systems (Munoz et al. 2019).

4 Propositional model counting

In this section, we provide some background for propositional logic, the #SAT problem, and different strategies employed by the evaluated solvers. Note that this section is not necessary to understand the empirical evaluation. Hence, the section can be skipped if considering the evaluated solvers as black boxes is sufficient for the reader.

Let F be a propositional formula and $vars(F)$ the corresponding set of variables with $|vars(F)| = n$. An assignment is a function $\alpha : vars(F) \rightarrow \{0, 1, undef\}$ that maps variables contained in F to the truth values (0 or 1) or undefined $(undef)$ (Kübler et al. 2010). Assignments can be partial, meaning that some variables $v \in vars(F)$ are mapped to $undef$. Otherwise, the assignment is called full (Kübler et al. 2010). For an assignment α, $|\alpha| \leq n$ corresponds to the number of variables mapped to 0 or 1 in α. We use $F(\alpha) \in \{0, 1\}$ to denote whether a full assignment α satisfies the formula F. We refer to assignments α with $F(\alpha) = 1$ as satisfying.

Propositional model counting (for short #SAT) is defined as the problem of computing the number of satisfying full assignments of a propositional formula (Gomes et al. 2006; Kübler et al. 2010). $\#F = |\{\alpha \mid F(\alpha) = 1\}|$ corresponds to the number of satisfying full assignments of formula F. In the following, we present three popular model counting methods employed by the majority of solvers in our empirical evaluation, namely (Davis-Putnam-Logemann-Loveland) DPLL-based (Bayardo Jr and Pehoushek 2000; Sang et al. 2005; Thurley 2006; Burchard et al. 2015; Biere 2008), d-DNNF-based (Darwiche 2004; Lagniez and Marquis 2017; Muise et al. 2010), and BDD-based (Toda and Soh 2016) counting.

The algorithms based on exhaustive DPLL iteratively assign variables to ultimately compute the number of satisfying assignments. The goal is to find an assignment that either satisfies or does not satisfy the formula for each possible assignment of the remaining $n - |\alpha|$ variables. If the formula evaluates to false under α, the number of resulting satisfying assignments for α is 0. If it evaluates to true, the number of satisfying assignments for α is $2^{n-|\alpha|}$, which is the number of possible assignments of the remaining variables. In particular, a satisfying full assignment induces exactly $2^{n-n} = 1$ solution. After computing a result for α, DPLL uses backtracking to find remaining assignments. The backtracking algorithm is performed until each satisfying assignment is covered. The sum of computed results is the exact number of satisfying assignments (Biere et al. 2009).

Another possible way to compute the number of satisfying assignments are d-DNNFs. The term d-DNNF stands for deterministic, decomposable negation normal form (Darwiche and Marquis 2002). A formula is in *negation normal form* (NNF) if the logical operators are limited to $\wedge$ (conjunction), $\vee$ (disjunctions), $\neg$ (negations) and negations only appear directly in front of literals (Huth and Ryan 2004). A formula F is called *deterministic* if each child $D_1, \ldots, D_n$ of a disjunction $D \in F$ is logically disjunct (i.e., $\forall i, j : i \neq j : D_i \wedge D_j \models \bot$) (Darwiche and Marquis 2002). Determinism implies that the children $D_1, \ldots, D_n$ of a disjunction D share no common solutions. Therefore, the number of satisfying assignments of the disjunction is equal to the sum of its children's results (i.e., $\#D = \sum_{i=1}^{n} \#D_i$) (Biere et al. 2009). A formula is called *decomposable* if the children $C_1, \ldots, C_n$ of a conjunction C share no variables (i.e., $\forall i, j : i \neq j : vars(C_i) \cap vars(C_j) = \emptyset$) (Darwiche and Marquis 2002). Decomposability implies that assignments for variables of the children $C_1, \ldots, C_n$ are independent of each other as the variables are disjoint. It follows that the number of satisfying assignments of the conjunction is equal to the product of the results for each child (i.e., $\#C = \prod_{i=1}^{n} \#C_i$) (Biere et al. 2009). Using both properties (determinism and decomposability), it is possible to compute the overall number of satisfying assignments by traversing the formula once (Biere et al. 2009). d-DNNF-based #SAT solving corresponds to compiling a propositional formula to d-DNNF and then retrieving the number of satisfying assignments by traversing the d-DNNF. After the compilation, computing the model count takes linear time w.r.t. the number of the d-DNNF nodes (Darwiche 2004).

Finally, #SAT may also be computed using a binary decision diagram $BDD(F)$ representing the propositional formula F. A binary decision diagram is a rooted directed acyclic graph two terminal nodes $\bot$ and $\top$. Every non-terminal node x is associated with a variable $v \in vars(F)$ and has precisely two outgoing edges, named *low* (setting v to false) and *high* (setting v to true). Typically, one considers reduced ordered binary decision diagrams (BDD) (Bryant 1986, 2018; Mendonça 2009). A BDD is *ordered*, if nodes associated to a variable v_i always precede nodes associated to a variable v_j or vice versa. If a BDD is *reduced*, then it (1) does not contain nodes with their low and high edges incident with the same node and (2) no two nodes associated with the same variable have the same nodes incident to their low and high edges. All satisfying assignments for F correspond to a path P from the root node to $\top$ (1-path) in $BDD(F)$. Let x_v be a node associated to the variable v. If the low edge of x_v is contained in P, then v is set to false. Analogously, v is set to true if P contains the high edge of x_v. If no node associated to v is contained in P, then v can be assigned an arbitrary value. Consequently, every 1-path induces $2^{n-|P|}$ satisfying assignments, where $|P|$ is the number of edges in P, and it suffices to iterate over all 1-paths in *BDD(F)* to compute $\#F$:

$$\#F = \sum_{\substack{\text{1-path } P \\ \text{in } \mathit{BDD(F)}}} 2^{n-|P|} \tag{1}$$

This can be achieved in linear time with respect to the number of nodes in $BDD(F)$ (Bryant 2018). BDDs are known to be sensitive to the order of variables and there are examples in which one order results in a BDD with linear number of nodes (w.r.t. the number of features) and another order results in a BDD of exponential size (Bryant 1986).

The main difference between the solving techniques is the reuse of results. DPLL-based solvers perform a single computation and typically just return the number of satisfying assignments (Bayardo Jr and Pehoushek 2000; Sang et al. 2004; Thurley 2006; Burchard et al. 2015; Biere 2008). When using a BDD or d-DNNF compiler, the resulting target format can be reused for further analysis. For example, d-DNNFs and BDDs can be used to compute the number of satisfying assignments under assumptions (Darwiche 2001; Heß et al. 2021), which could be used to compute the number of valid configurations containing a certain feature or an arbitrary combination of features (i.e., a partial configuration). Thus, if multiple #SAT computations are required, compiling into d-DNNF or BDD might be beneficial even if the compilation time takes longer than a DPLL-based computation.

5 Experiment design

In this section, we present the experiment design for our evaluation of #SAT solvers on industrial feature models. We provide the required information for the presentation (Section 6) and discussion (Section 7) of the results. The replication package for our empirical evaluation is publicly available.[2].

In Section 5.1, we explain the research questions we aim to answer in our evaluation. In Section 5.2, we present the gathered #SAT solvers and the methodology we used to collect them. In Section 5.3, we discuss the selection of subject feature models and provide information (e.g., number of features and domain) of the underlying product line. In Section 5.4, we describe the setup of the experiments regarding the overall procedure of the measurements, considered solvers, considered systems, and applied statistical tests. In Section 5.5, we provide details on the technical setup for the evaluation.

5.1 Research questions

In this section, we discuss the research questions that we aim to answer with the empirical evaluation. The research questions provide insight on the general scalability of #SAT technology, the performance of exact #SAT solvers and solver classes, the correlation between structural metrics of the feature model and the runtime of solvers, and the performance of approximate #SAT solvers. Typically, feature-model analyses, such as counting, are applied in interactive settings or in continuous integration. As those settings mandate short runtimes, we consider an analysis to be scalable if it requires at most a few minutes of runtimes.

RQ1 How do #SAT solvers perform on industrial feature models?

To use applications based on the feature-model cardinality in industry, we need to identify solvers that scale for the task of analyzing industrial feature models. Thus, we examine the performance of exact #SAT solvers regarding the runtime when computing the cardinality of industrial feature models. Furthermore, we aim to find the most efficient #SAT solvers to provide recommendations on which #SAT solvers to use. Here, we consider the runtime required to compute #SAT for given feature models as the efficiency of a solver.

RQ2 How do different classes of #SAT solvers perform on industrial feature models?

We consider multiple classes of #SAT solvers (cf. Section 5.3). With RQ2, we analyze the runtimes of different categories of solvers, namely (1) DPLLbased, (2) algebraic-based, and knowledge compilers translating to (3) BDD, (4) d-DNNF, and (5) other formats (i.e., EADT and SDD).We use the insights to discuss the benefits of different solver categories and to give recommendations on promising techniques for counting-based analyses.

RQ3 How does the runtime of #SAT solvers correlate to structural metrics of the feature model?

[2] https://doi.org/10.5281/zenodo.7329979

We aim to provide some first insights on which properties cause a feature model to be hard to analyze for #SAT solvers. In particular, we examine if there is a correlation between the performance of the #SAT solvers and structural metrics related to the size and complexity of feature models. The insights can be used as an indicator on the scalability of existing #SAT solvers for a given feature model depending on their structure. Furthermore, we identify metrics that have a high impact on performance to find promising candidates for more accurate performance predictions in the future. In Table 1, we provide a list of metrics we examined. For each metric, we provide a description and instructions on how to compute the respective metric for a given feature model. The metrics were collected by Bezerra et al (Bezerra et al. 2014) and Bagheri and Gasevic (Bagheri and Gasevic 2011). Those metrics are based on structural properties related to size (e.g., number of features) or related to complexity (e.g., cyclomatic complexity).

RQ4 How do approximate #SAT solvers perform on industrial feature models?

In addition to exact #SAT solvers, we examine the performance of approximate #SAT solvers which estimate the number of satisfying assignments for a given formula. There are applications which require exact results, such as uniform random sampling where approximate results would violate uniformity. However, for a multitude of applications, an estimated cardinality may be often sufficient. For example, consider prioritizing features that appear in many valid configurations for two features A and B with $\#A = 10^{65}$ and $\#B = 10^{60}$. For instance, an approximation that ensures that the result is at most 20 times larger/smaller than the exact count, would result in the same prioritizations as exact results as $\frac{1}{20} \cdot 10^{65} > 20 \cdot 10^{60}$. We examine the performance of approximate #SAT solvers to provide insights on their benefits.

5.2 Evaluated #SAT solvers

In the following, we present the #SAT solvers used in our empirical evaluation. First, we describe our methodology of gathering the solvers. Second, we list the identified solvers, group them by their type of computing #SAT, and provide pointers where to find them.

Methodology Our main goal for selecting the solvers is a representative coverage of publicly available #SAT solvers. Such a coverage should allow for conclusive results on (1) the current scalability of #SAT technologies when applied to feature models and (2) recommendations on which solvers should be used to analyze feature models at hand.

We included all solvers that satisfy the following criteria: First, the solver needs to be publicly available (i.e., source code or binary is provided at generally accessible URL). Second, the tools have to accept CNFs in DIMACS format as input. DIMACS is the de facto standard for representing CNFs and used by the vast majority of SAT and #SAT solvers (Biere 2008; Bayardo Jr and Pehoushek 2000; Sang et al. 2004, 2005; Darwiche 2002, 2004; Toda and Soh 2016). Third, the solver can be used as a standalone blackbox tool in contrast to tools that require further setup (e.g., a client-server architecture (Lagniez et al. 2018)). Furthermore, we excluded the tool GPUSAT (Fichte et al. 2019) which performs #SAT on a GPU as we expect issues with the comparability if a solver uses different hardware. We identified #SAT solvers with the following three approaches.

First, we collected work that performs counting-related analyses on product lines (Kübler et al. 2010; Munoz et al. 2019; Oh et al. 2016; Pérez Morago 2016; Heradio-Gil et al. 2011; Fernández-Amorós et al. 2014; Chen and Erwig 2011; Pohl et al. 2011) to identify #SAT tools and respective publications used in product-line analysis. Then, we performed

Table 1 Structural feature-model metrics (**RQ3**)

Number of features	
Description	Number of features in the feature model overall
Formula	$\|Features\|$, with $Features$ being the set of features
Number of leaf features	
Description	Number of features in the feature model without children
Formula	$\|Leaves\|$, with $Leaves$ being the set of leaf features
Number of top features	
Description	Number of children of the root feature
Formula	$\|Top\|$, with Top being the set of children of the root feature
Number of cross-tree constraints	
Description	Number of cross-tree constraints in the feature model
Formula	$\|CTC\|$, with CTC being the set of cross-tree constraints
Number of clauses	
Description	Number of clauses in the CNF representing the feature model
Formula	$\|Clauses\|$, with $Clauses$ being the set of clauses in the CNF
Number of literals	
Description	Number of overall literals appearing in the CNF
Formula	$\|Literals\|$, with $Literals$ being the set of literals in the CNF
CTC-Density	
Description	Ratio of unique features appearing in CTC compared to overall number of features
Formula	$\frac{\|Features_{CTC}\|}{\text{\#Features}}$, with $Features_{CTC}$ being the set of features appearing in a cross-tree constraint.
Depth of tree	
Description	Depth of the feature tree at the longest path
Formula	$\|Features_{LP}\|$, with $Features_{LP}$ being the set of features in the longest path from the root to a leaf feature.
Flexibility of configuration	
Description	Ratio of optional features compared to overall number of features
Formula	$\frac{\|Features_{Opt}\|}{\|Features\|}$ with $Features_{Opt}$ being the set of features that appear in some but not all valid configurations.
Ratio of variability	
Description	Average number of children
Formula	$\frac{\sum_{f \in Features} \|children_f\|}{\|Features \setminus Leaves\|}$, with $children_f$ being the set of children of feature f.
Coefficient of connectivity-density	
Description	Number of edges between features compared to the number of features
Formula	$\frac{\sum_{f \in Features} \|edges_f\|}{2 \cdot \|Features\|}$, with $edges_f$ being the set of distinct edges connecting features. For a clause $(f_1 \vee \ldots \vee f_n)$, there is an edge between every pair of feature $f_1, \ldots, f_n$.
Cyclomatic complexity	
Description	Number of distinct cycles between features
Formula	$\|cycles_f\|$ with $cycles_f$ being the set of independent cycles spanned by $edges_f$

(forward and backward) snowballing from the identified publications. For backward snowballing, we employed data from Google Scholar. Second, we used a list of publicly available #SAT solvers from the report of the model counting 2020 competition as comparison (Fichte

Table 2 Overview exact #SAT solvers

Solver	Type	Target format	Reference
Cachet	DPLL	–	(Sang et al. 2004, 2005)
countAntom	DPLL	–	(Burchard et al. 2015)
Ganak	DPLL	–	(Sharma et al. 2019)
PicoSAT	DPLL	–	(Biere 2008)
Relsat	DPLL	–	(Bayardo Jr and Pehoushek 2000)
SharpCDCL	DPLL	–	(Klebanov et al. 2013)
sharpSAT	DPLL	–	(Thurley 2006)
McTW	Algebraic	–	MC Competition (Fichte et al. 2021)
SUMC1	Algebraic	–	MC Competition (Fichte et al. 2021)
ADDMC	Algebraic	–	MC Competition (Fichte et al. 2021)
c2d	Compiler	d-DNNF	(Darwiche 2002; 2004)
d4	Compiler	d-DNNF	(Lagniez and Marquis 2017)
dSharp	Compiler	d-DNNF	(Muise et al. 2010)
BuDDy	Compiler	BDD	http://buddy.sourceforge.net/manual/main.html. Accessed: 02 Mar 2020
CNF2OBDD	Compiler	BDD	(Toda and Soh 2016)
Cudd	Compiler	BDD	https://github.com/vscosta/cudd. Accessed: 13 Jun 2020
CNF2EADT	Compiler	EADT	(Koriche et al. 2013)
MiniC2D	Compiler	SDD	(Oztok and Darwiche 2015)
SDD	Compiler	SDD	(Darwiche 2011)

et al. 2021). Both lists are similar with eleven shared #SAT solvers. The list of the model counting competition contained one solver in addition to the eleven shared solvers while the list from product-line analysis contained four additional solvers. Due to the similarity in the identified solvers, we argue that our list of #SAT solvers provides a reasonable representation of current #SAT technology. Third, we added three #SAT solvers that entered the model counting 2020 competition but were not published beforehand.[3]

Selected solvers Overall, we gathered 19 exact and 2 approximate #SAT solvers. Table 2 provides an overview on the exact solvers. The exact #SAT solvers can be separated in three main categories: DPLL-based solvers, algebraic solvers, and knowledge compilers. We consider a knowledge compiler to be a #SAT solver if the compiled target language and the compiler support computing the number of satisfying assignments in polynomial time in the size of the target formula.

The solver countAntom is the only solver which internally supports multi-threading (Burchard et al. 2015). We evaluated countAntom with one and four available threads separately to examine the impact of multi-threading on the runtime. We consider evaluating countAntom also with four threads (opposed to only with a single thread) as the more sensible option due to the following reasons: First, it is reasonable to assume that multiple threads would be used in industrial settings. Second, to allow multi-threading the developers

[3] https://doi.org/10.5281/zenodo.4292581

of countAntom made several adjustments which may put countAntom at a disadvantage when using a single thread. Third, nevertheless, a larger number of threads may result in a too large advantage for countAntom. During the remainder of the evaluation, we refer to countAntom with four threads if not stated otherwise.

Neither BuDDy http://buddy.sourceforge.net/manual/main.html. Accessed: 02 Mar 2020 nor CUDD https://github.com/vscosta/cudd. Accessed: 13 Jun 2020 support parsing DIMACS directly. In previous work, we implemented a Python-based wrapper called ddueruem[4] (Heß et al. 2021) which uses the ctypes library[5] to interface with their shared libraries and construct the BDD using the API described in their respective manuals http://buddy.sourceforge.net/manual/main.html. Accessed: 02 Mar 2020; https://github.com/vscosta/cudd. Accessed: 13 Jun 2020. As suggested in the manuals of BuDDy and CUDD, we enabled automatic variable reordering in both BuDDy and Cudd, using the converging variant of the sift algorithm (Rudell 1993). We decided to use our own wrapper ddueruem due to limitations, namely the missing support for Cudd 3.0.0 and frequent crashes when using BuDDy in the JavaBDD[6] framework, which is often used for product-line analysis (Mendonça 2009; Mendonca and Cowan 2010; Pohl et al. 2011).

In addition to the exact #SAT solvers, we also identified two approximate #SAT solvers, namely ApproxMC (Chakraborty et al. 2013) and ApproxCount (Wei and Selman 2005). The solver ApproxCount iteratively assigns variables to reduce the complexity of a formula. For each assigned variable, the solver estimates the resulting reduction in the number of satisfying assignments. After a user-specified number of assigned variables, the exact #SAT solver Cachet is executed with the simplified formula as input. The estimated reduction is then applied to the result of the simplified formula to derive an approximated number of satisfying assignments for the original formula. In our previous work (Sundermann et al. 2020), every feature model with less than 1,000 features was successfully evaluated by most #SAT solvers. Following this insight, we directed ApproxCount to start the exact computation at 1,000 remaining variables. For ApproxMC, we used the default parameters.

5.3 Subject systems

The main goal of the empirical evaluation is to examine the applicability of #SAT solvers for analyzing feature models. We argue that the applicability mainly depends on the scalability on industrial feature models, as artificial models might not be representative for industrial usage as observed in other domains (Ansótegui et al. 2009; Heß et al. 2021). Therefore, we only use industrial feature models as subject systems. We consider a feature model to be industrial if it fulfills the following two criteria: (1) it specifies the variability of a product line used in the real world and (2) it does not vastly simplify the complexity (in terms of features and constraints) of the product line. Note that we only consider variability of the problem space (i.e., which valid configurations do exist) opposed to variability in the solution space (i.e., how and where to implement variability).

Selected systems With our selection of subject systems, we aim for a wide coverage of different domains. We evaluate the performance of the listed #SAT solvers on feature models taken from industrial product lines from the automotive, operating system, database, and

[4] https://github.com/SoftVarE-Group/emse-evaluation-sharpsat/tree/v1.0/solvers/ddueruem

[5] https://docs.python.org/3/library/ctypes.html

[6] http://javabdd.sourceforge.net/

Table 3 Overview Subject Systems (Sorted by #Features)

Subject systems	i	#Features	#Constraints	Orig. source
BerkeleyDB	1	76	20	https://github.com/FeatureIDE/FeatureIDE
axTLS	2	96	14	(Knüppel et al. 2017)
uClibc	3	313	56	(Knüppel et al. 2017)
uClinux-base	4	380	3,455	(Knüppel et al. 2017)
Automotive04	5	531	623	Confidential
Automotive03	6	588	1,184	Confidential
BusyBox	7	631	681	(Pett et al. 2021)
FinancialServices	8	771	1,080	(Nieke et al. 2018; Pett et al. 2019)
Embtoolkit	9	1,179	323	(Knüppel et al. 2017)
CDL (116 Models)	10	1,178–1,408	816–956	(Knüppel et al. 2017)
uClinux-distribution	11	1,580	197	(Knüppel et al. 2017)
Automotive05	12	1,663	10,321	Confidential
Automotive01	13	2,513	2,833	https://github.com/FeatureIDE/FeatureIDE
Linuxv2.6.33.3	14	6,467	3,545	(Knüppel et al. 2017)
Automotive02	15	18,616	1,369	(Knüppel et al. 2017)

financial services domain. Table 3 provides an overview on the considered feature models, sorted by the number of features, including name, number of features, number of constraints, and the work they were originally published in. The index *i* indicates the position of the subject system in diagrams in Section 6.

First, we analyze feature models provided by Knüppel et al (Knüppel et al. 2017).[7] The authors extracted the systems from snapshots of an automotive product line and by translating KConfig and CDL models. KConfig[8] is a language designed for managing Linux configurations and CDL for managing eCos[9], a configurable operating system for embedded applications (Knüppel et al. 2017). The considered KConfig models are *axTLS*, *uClibc*, *uClinux-base*, *Embtoolkit*, *uClinux-distribution*, and *Linux*. In addition, Knüppel et al provide an automotive product line *Automotive02*. Second, we evaluate the solvers on *BusyBox* provided by Pett et al (Pett et al. 2021).[10] Third, we include a feature model from the *FinancialServices* domain (Nieke et al. 2018; Pett et al. 2019). Fourth, we consider the systems *Automotive01* (Kowal et al. 2016) and *BerkeleyDB* (Kästner et al. 2007) which are available as FeatureIDE examples.[11] Fourth, we were given access to industrial models for three different systems from the automotive domain (*Automotive03-Automotive05*). These models were provided in a proprietary format. With the help of company interns, we translated their configuration knowledge into feature models. For our entire experiment, we translated each feature model to the DIMACS format using FeatureIDE 3.5.5.[12]

For some subject systems, namely *Automotive02–05*, *FinancialServices*, and *BusyBox* a history of feature models is available, each representing a unique timestamp. For each

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

[8] https://www.kernel.org/doc/html/latest/kbuild/kconfig-language.html

[9] https://ecos.sourceware.org/

[10] https://github.com/TUBS-ISF/Stability-of-Productline-Sampling

[11] https://github.com/FeatureIDE/FeatureIDE

[12] https://github.com/FeatureIDE/FeatureIDE/releases/tag/v3.5.5

feature model with a history, we consider only the latest version. Thoroughly analyzing the entire history is out of scope and left as future work.

In previous experiments (Sundermann et al. 2020), we found that the 116 CDL models are highly similar regarding a variety of metrics (i.e., all metrics considered Section 5.1) and also resulted in similar runtimes for #SAT solvers. If we evaluated the different CDL models as distinct systems, 89.2% of the overall 130 (14 other + 116 CDL) evaluated feature models are CDL models which results in a huge bias of the results. Therefore, we consider the median of runtimes over the 116 different CDL models as the runtime of the CDL subject system in every experiment if not stated otherwise. To show the similarity in the performance of #SAT solvers, we also present the runtimes on the different CDL models in Section 6.

5.4 Experimental setup

In this section, we describe the procedure of the experiments conducted to gather insights to answer our research questions.

To analyze the feature models with #SAT solvers, we translate each feature model into conjunctive normal form (CNF) and store it in DIMACS format. We invoke the #SAT solvers with the DIMACS as input. As the translation to CNF typically requires only a few milliseconds and is equivalent for each solver, we do not include the translation time in the overall runtime. Furthermore, we set a timeout of ten minutes (cf. **RQ1** in Section 5.1) for evaluating a single feature model as the baseline for the experiment. The threshold is motivated by applying counting-based analyses in interactive settings and continuous-integration environment which should not exceed a few minutes of runtime to provide fast feedback to developers after changing a feature model.

An important aspect we consider for our benchmark is the trade-off between significance of results and ecological footprint. If a solver hits the timeout of ten minutes for every single one of the 130 feature models, the evaluation would require almost a day of continuous runtime considering a single repetition. Performing a number of repetitions that allows for significant results would require substantially more runtime. For instance, when using 50 repetitions this would potentially result in more than 11 years of nonstop computation time. Thus, we aim to reduce the overall runtime of the experiments while preserving significant results.

In the following, we explain the performed experiments in detail. Table 4 provides an overview over the two experiments regarding considered solvers, considered research questions, and number of performed repetitions per measurement.

Table 4 Overview experiments

Experiment	Solvers	#Reps.	RQ1	RQ2	RQ3	RQ4
Experiment 1a	All Exact #SAT	1	×	×		
Experiment 1b	Remaining Exact #SAT	50	×	×	×	
Experiment 1c	Remaining Exact #SAT	1	×	×		
Experiment 2a	All Approximate #SAT	1				×
Experiment 2b	Remaining Approximate #SAT	50				×

Experiment 1: Scalability of exact #SAT solvers In the first experiment, we measure the runtimes of the exact #SAT solvers (cf. Section 5.2) on the considered feature models (cf. Section 5.3). For the solvers based on knowledge compilation, we consider the overall runtime to compile and compute a result. We use the insights of this experiment to answer **RQ1**, **RQ2**, and **RQ3**. The experiment is separated into three stages.

In the first stage (Experiment 1a), we identify and filter slow #SAT solvers. Here, we measure the runtime of each of the 19 exact #SAT solvers on the 15 subject systems. The idea of Experiment 1a is to remove slow solvers from the following two stages that significantly increase the overall runtime for the experiments. We consider a solver to be slow if the solver requires more than 50% of additional runtime compared to the following solver when ordered by overall runtime for the experiment. We refer to the solvers that are not excluded as *remaining* solvers. In the second stage (Experiment 1b), we perform the measurements with the remaining #SAT solvers with 50 repetitions for each feature model for more robust results. In the third stage (Experiment 1c), we further evaluate the runtimes of the remaining solvers on subject systems for which no solver computed a result in Experiment 1a and 1b. Here, we perform one repetition and increase the timeout to 24 hours to examine whether an increase of the timeout allows a successful computation.

Orthogonal to the measurements of Experiment 1a, 1b, and 1c, we examine the number of valid configurations for the considered feature models. Here, we use the results computed by the solvers.

Experiment 2: Scalability of approximate #SAT solvers In the second experiment, we examine the scalability of the two considered approximate #SAT solvers on each feature model to provide insights for **RQ4**. Furthermore, we give a comparison to the exact #SAT solvers to evaluate the benefits. Analogous to Experiment 1a, we also perform an initial experiment referred to as Experiment 2a with only one repetition per measurement to exclude slow solvers from the following experiments. Then, we repeat the measurements with 50 repetitions on the remaining approximate #SAT solvers (Experiment 2b), analogous to Experiment 1b.

Statistical tests We apply the following statistical tests to evaluate the significance of our results depending on the use case. Table 5 gives an overview on the use cases, tests we used for each use case, and the **RQs** that are dependent on the given use case.

For the first use case *Comparison Solvers System*, we compare the performance of different solvers on each of the feature models separately. For the comparison, we consider the 50 repetitions of a solver/system combination as sample. Here, we apply a Mann-Whitney significance test (McKnight and Najab 2010) as we have *unpaired* samples and do not assume

Table 5 Overview statistical tests

Use case	Sample	Statistical tests	RQ1	RQ2	RQ3	RQ4
Comparison solvers system	Unpaired	Mann-Whitney (McKnight and Najab 2010)	×	×		×
Comparison solvers overall	Paired	Friedman test (Conover and Iman 1981) Post-Hoc Conover (Conover and Iman 1981)	×	×		×
Correlation solver/metric	Paired	Spearman (Zar 1972)			×	

Table 6 Spearman: levels of correlation

Correlation	Value range
Very weak	$0 \leq \|r_s\| < 0.2$
Weak	$0.2 \leq \|r_s\| < 0.4$
Moderate	$0.4 \leq \|r_s\| < 0.6$
Strong	$0.6 \leq \|r_s\| < 0.8$
Very strong	$0.8 \leq \|r_s\| \leq 1.0$

a normal distribution. For the tests, we assume the typical significance level of $\alpha = 5\%$. We use the `scipy`v1.7.2 implementation of Mann-Whitney.[13]

For the second use case *Comparison Solvers Overall*, we compare the overall performance of the different #SAT solvers on all 15 subject systems. For the comparison, each data point corresponds to the median of runtimes over the 50 repetitions for a combination of subject system and solver. Here, we apply a Friedman Test followed by a Post-hoc Conover test on the samples of all solvers as we have *paired* samples (pairs of subject systems) and again do not assume a normal distribution (Conover and Iman 1981). For the tests, we assume a significance level $\alpha = 5\%$. We use the `scipy`v1.7.2 implementation of Friedman[14] and the `scikit_posthocs` implementation of Post-Hoc Conover.[15]

For the third use case *Correlation Solver/Metric*, we evaluate the correlation between the runtime of #SAT solvers and structural metrics (cf. **RQ3**). Here, we use Spearman's correlation coefficient r_s (Zar 1972) to evaluate the strength of the correlation. For two variables, r_s describes their correlation with a value between -1 and 1. The values 1 and -1 describes a very strong positive or negative correlation, respectively. $r_s = 0$ indicates that the variables have no correlation at all. We expect that Spearman's coefficient provides more sensitive results (compared to Pearson's coefficient) due to the following reasons (Artusi et al. 2002). First, Spearman's coefficient is suitable to detect non-linear relationships between the variables, and it is possible that the correlation between a metric and the runtimes is not linear. Second, there may be significant outliers which tend to cause problems for the expressiveness of Pearson's coefficient. Table 6 shows the levels of correlation strength for the Spearman's coefficient r_s we use. We use the `scipy`v1.7.2 implementation of Spearman to compute the correlation coefficients[16].

In addition to significance tests, we compute effect sizes (Sullivan and Feinn 2012) for samples shown to be significantly different. In particular, we employ Cohen's d which describes the difference between the median of two samples relative to the standard deviation (Sullivan and Feinn 2012). Table 7 shows the levels of effect sizes with their range of d values (Sullivan and Feinn 2012).

5.5 Technical setup

Each experiment was performed on a *Linux CentOS 8* system with *64*-bit architecture. The evaluated machine uses an *Intel Core Broadwell Processor* that consists of *16* sockets with

[13] https://docs.scipy.org/doc/scipy/reference/generated/scipy.stats.mannwhitneyu.html

[14] https://docs.scipy.org/doc/scipy/reference/generated/scipy.stats.friedmanchisquare.html

[15] https://scikit-posthocs.readthedocs.io/en/latest/generated/scikit_posthocs.posthoc_conover/

[16] https://docs.scipy.org/doc/scipy/reference/generated/scipy.stats.spearmanr.html

Table 7 Cohen: levels of effect size

Effect size	Value range
Very small	$0 \leq d < 0.2$
Small	$0.2 \leq d < 0.5$
Medium	$0.5 \leq d < 0.8$
Large	$0.8 \leq d < 1.3$
Very large	$1.3 \leq d$

one core each. The clock rate is *2,394 Mhz* and the machine contains *62 GB* of RAM. For each computation and experiment, we limit the memory usage to 8 GB due to the following two reasons. First, we assume that a memory limit of 8 GB reflects the capacity for RAM usage on common PCs or notebooks. Second, in preliminary experiments, we found that further increasing the memory limit yields little to no benefits for the runtimes of #SAT solvers. For the measurements, we implemented a *Python* framework to (1) call the solver binaries and provide the input, (2) measure the runtimes with the *timeit* module,[17] and (3) limit the memory usage. For reproducibility, the framework, solvers, and input data are publicly available.[18] *Hyper-threading*, *turboboost*, and *caching of the file system* were disabled during the entire measurements to reduce computational bias. Furthermore, no other major computations were run on the system during the experiments.

6 Results

In this section, we present the results of our empirical evaluation separated into the two presented experiments.

6.1 Experiment one: Exact #SAT solvers

Experiment 1a Figure 2 shows the runtime of all exact #SAT solvers on each subject system. Each point on the x-axis corresponds to one of the 15 subject systems. The systems are sorted by the number of features in ascending order (cf. Table 3). The y-axis shows the runtime of the different solvers with a logarithmic scale. The different categories are indicated by the colors of the markers (orange = DPLL, purple = algebraic, green = d-DNNF, blue = BDD, gray = knowledge compilers to other formats). The red line indicates that a solver hits the timeout. The blue line indicates that an error occurred or a solver passed the memory limit. *CDL Median* corresponds to the median over all 116 CDL feature models (cf. Section 5.3). The majority of systems (13/15) was successfully evaluated within 10 minutes by at least one solver. For each of the 13 solved systems, the fastest solver required less than one second. However, none of the solvers was able to compute the cardinality of the other two systems, namely *Automotive05* and *Linux*.

Figure 3 shows the sum of runtimes of each evaluated exact #SAT solver on all 15 subject systems in Experiment 1a. Each bar corresponds to the sum of runtimes for one solver. Note that this sum only includes the median of runtimes for the 116 CDL models instead of the

[17] https://docs.python.org/3/library/timeit.html

[18] https://doi.org/10.5281/zenodo.7329979

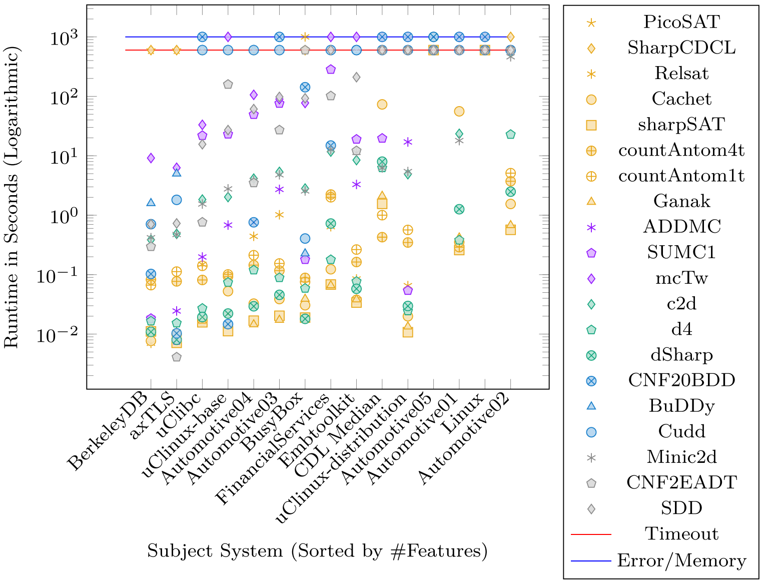

Fig. 2 Runtime in seconds for all exact #SAT solvers

overall sum (cf. Section 5.3). Considering a timeout of 10 minutes per system, the maximum runtime is 150 minutes (hitting the timeout for all 15 systems) which is indicated by the red line. The solvers are sorted by the overall sum of runtimes (ascending). If a subject system could not be evaluated due to timeout, memory limit, or an arbitrary error, we added the timeout (10 minutes) to the overall runtime. Table 8 gives an overview of the performance

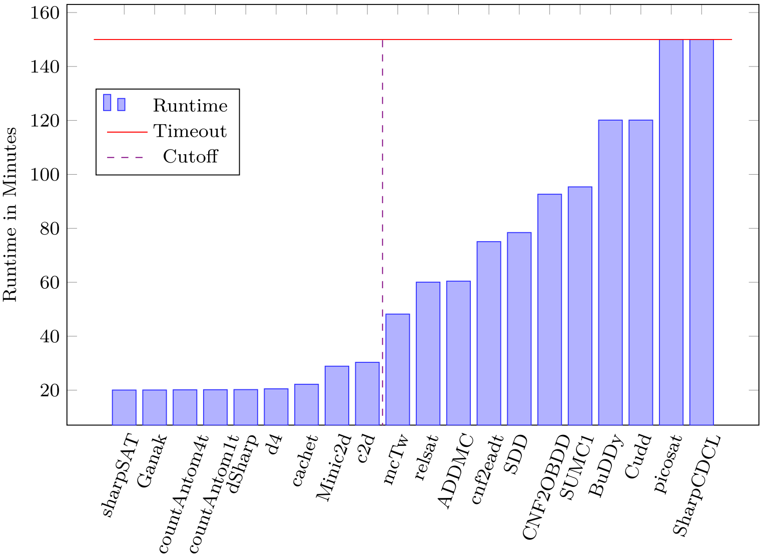

Fig. 3 Runtime of all solvers to evaluate the 15 subject systems

Table 8 Overview Experiment 1a

Solver	Solved	% Solved	Ov. Runtime (s)	Remaining
sharpSAT	13	87	1,202.6	✓
Ganak	13	87	1,203.4	✓
countAntom4t	13	87	1,207.8	✓
countAntom1t	13	87	1,210.2	✓
dSharp	13	87	1,212.7	✓
d4	13	87	1,230.1	✓
Cachet	13	87	1,339.9	✓
miniC2D	13	87	1,733.6	✓
c2d	13	87	1,818.9	✓
mcTw	11	73	2,892.0	✗
Relsat	9	60	3,602.4	✗
ADDMC	9	60	3,625.0	✗
CNF2EADT	8	53	4,504.1	✗
SDD	8	53	4,705.7	✗
CNF2OBDD	6	33	5,558.1	✗
SUMC1	6	33	5,721.8	✗
BuDDy	3	20	7,206.6	✗
Cudd	3	20	7,206.8	✗
SharpCDCL	0	0	9,000.0	✗
PicoSAT	0	0	9,000.0	✗

for the different solvers. Eight of the solvers, namely `sharpSAT`, `Ganak`, `countAntom` (both with four and one thread), `d4`, `Cachet`, `dSharp`, `MiniC2D`, and `c2d` evaluated 13 out of 15 (86.7%) subject systems within eleven minutes of runtime. The eleven slower solvers, namely `McTW`, `Relsat`, `ADDMC`, `CNF2EADT`, `SDD`, `CNF2OBDD`, `SUMC1`, `BuDDy`, `Cudd`, `SharpCDCL`, and `PicoSAT`, successfully evaluated at most 73.3% of the 15 subject systems with a timeout of ten minutes for each model. Furthermore, the fastest of the slower solvers (`McTW`) requires around 60% more runtime than the slowest of the faster solvers (`c2d`). Overall, the eleven slower solvers took 96.8% of the total runtime (10.2 days) required for Experiment 1a. Performing the 50 repetitions with all solvers would result in around 1.4 years of continuous computation just for Experiment 1b. For all following experiments, we only include the eight fastest #SAT solvers. In Table 8 the excluded solvers are marked with an ✗-mark in the column *remaining*. In Fig. 3, the dashed violet line marks the cut for the excluded solvers. Each solver on right side of the line is excluded from the following experiments.

Each BDD-based #SAT solver successfully evaluated at most 6 of the 15 systems and required at least 92 minutes overall. Every d-DNNF-based solver needed less than 31 minutes for all subject systems and only failed to evaluate *Linux* and *Automotive05*.

Figure 4 shows the runtime of the 19 exact #SAT solvers for the 116 CDL feature models. Each point on the x-axis corresponds to one CDL model. The y-axis shows the runtime of different solvers in seconds with a logarithmic scale. For all solvers but `Cachet`, the median runtime over all 116 feature models is smaller than two times the minimum value (i.e., the shortest runtime required for one of the 116 feature models for that solver). Also,

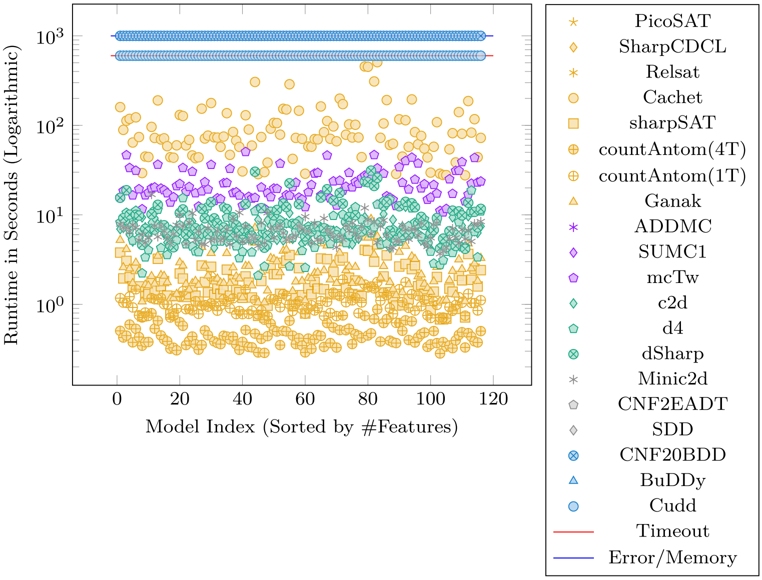

Fig. 4 Runtime in seconds for all exact #SAT solvers on CDL models

the maximum is always smaller than two times the median. The results support our claim in Section 5.3 that CDL models are highly similar and handling them as separate subject systems would result in a bias of the measured runtimes.

Experiment 1b In Experiment 1b, we measured each combination of the feature models and the eight remaining solvers with 50 repetitions for more reliable results (cf. Section 5.4). Figure 5 shows the median runtimes and standard deviation for each solver-system combination. In the remainder of this section, we only consider the 13 systems successfully evaluated by at least one of the solvers if not stated otherwise. Considering the overall sum of runtimes, the three solvers requiring the least runtime are `sharpSAT` (2.5 seconds), `Ganak` (3.3 seconds), and `countAntom` (7.8 seconds). Over the 13 systems, `sharpSAT` is significantly ($p < 0.03$) faster than every #SAT solver but `Ganak`, `Cachet`, and `dSharp`. However, each effect size is small ($d < 0.47$) which matches the expectations as the large differences in runtime between the subject systems for each solver result in a large standard deviation.

`sharpSAT` is significantly ($p < 0.004$) faster with mostly (88.1%) very large effect sizes ($d > 1.53$) than all other solvers on 6 of the 13 systems. `Ganak` is significantly ($p < 10^{-11}$) faster than all other #SAT solvers with very large effect sizes ($d > 1.61$) for *Automotive03*. `countAntom` is significantly faster ($p < 10^{-17}$) than all other solvers with very large effect sizes ($d > 1.73$) on all 116 *CDL* models but for no other system. `Cachet` is significantly ($p < 10^{-7}$) faster than all other solvers for three smaller (less than 1,200 features) systems, namely *axTLS*, *embToolkit*, and *BerkeleyDB* with all effect sizes being very large ($d > 1.38$) but one with a medium effect size ($d = 0.78$).

We also compared `countAntom` with four and one thread. `countAntom` with four threads is significantly ($p = 0.028$) faster than with one thread for the 13 systems overall. Still, `countAntom` with one thread is significantly ($p < 0.036$) faster for three subject systems, namely *embtoolkit* ($d = 0.60$), *uClinux-base* ($d = 0.82$), and *Automotive03*

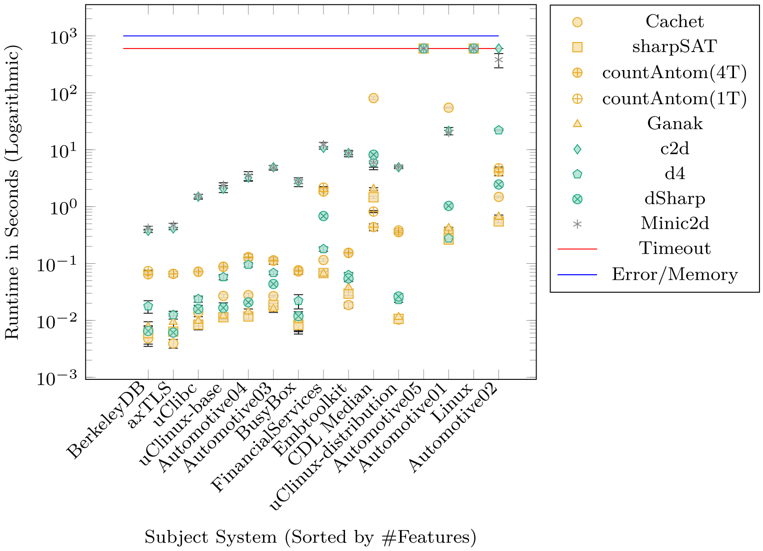

Fig. 5 Runtime (median & standard deviation) in seconds for remaining exact #SAT solvers

($d = 0.21$). Overall, `countAntom` with four and one thread require 7.8 and 9.2 seconds of runtime, respectively.

Table 9 shows the correlation between structural metrics of the feature models and the runtime of #SAT solvers. 'Correlation Fastest' shows the correlation between each metric and the runtime of the fastest solver for each instance. Note that the fastest solver varies depending on the evaluated feature model. There is a very strong positive ($r_s > 0.8$) correlation between the runtime and the following metrics: number of features, number of leaf features, number of cross-tree constraints, number of literals, and number of clauses. Consequently, for instance, the runtime of #SAT solvers tends to increase if the number of features increases. Also, there is a strong correlation between the cyclomatic complexity and the runtime. Every other metric correlates either weakly (0.2–0.39) or very weakly ($r_s < 0.2$) with the runtime of the fastest solver. 'Correlation Range' shows the minimum and maximum correlation between a metric and a solver. For every metric that has a strong correlation with the runtime of the fastest solver, each solver has an at least strong correlation. This observation is analogous for weakly correlated metrics with one exception (`countAntom` has a moderate correlation with the connectivity density).

Figures 6 and 7 show the runtime of the eight remaining solvers in relation to the number of features and the number of constraints, respectively. In each diagram, both scales are logarithmic. Every system with either fewer than 1,000 features or 1,000 constraints was evaluated within 0.5 seconds. While there is a strong correlation between the runtimes of the #SAT solvers and both metrics (i.e., number of features and constraints), a feature model with respectively more features or constraints does not guarantee a longer runtime. The two systems that reached the timeout, namely *Linux* and *Automotive05*, contain 6,467 and 1,663 features. *Automotive02* which contains 18,616 features was evaluated within 0.5 seconds. It is important to note that *Automotive02* contains only 1,369 constraints while *Linux* and *Automotive05* contain 3,545 and 10,321 constraints, respectively. Also, *uClinux-base* contains 3,455 constraints, but the fastest solver is about 50 times faster than for *Automotive02*.

Table 9 Correlation between structural metrics and runtime of #SAT solvers

Metric	Coefficient fastest	Coefficient range
Number of literals	0.89 (very strong)	0.82 (very strong)–0.89 (very strong)
Number of clauses	0.87 (very strong)	0.80 (very strong)–0.87 (very strong)
Number of features	0.85 (very strong)	0.73 (strong) –0.94 (very strong)
Number of leaf features	0.82 (very strong)	0.68 (strong) –0.92 (very strong)
Number of constraints	0.81 (very strong)	0.67 (strong)–0.84 (very strong)
Cyclomatic complexity	0.77 (strong)	0.64 (strong)–0.79 (strong)
Tree depth	0.34 (weak)	0.29 (weak)–0.39 (weak)
Connectivity density	0.20 (weak)	0.11 (very weak)–0.57 (moderate)
Ratio of variability	0.11 (very weak)	-0.01 (very weak)–0.27 (weak)
Number of top features	0.06 (very weak)	-0.01 (very weak)–0.17 (very weak)
Flexibility of configuration	0.07 (very weak)	-0.01 (very weak)–0.18 (very weak)

Experiment 1c In Experiment 1c, we invoked the remaining solvers with a timeout of 24 hours for the two systems which hit the timeout for every solver in every repetition, namely *Automotive05* and *Linux*. Neither of the remaining eight solvers was able to compute the cardinality for either system within 24 hours.

Feature-model cardinalities Table 10 shows the cardinalities (i.e., the number of valid configurations) of the evaluated subject systems. The systems are sorted by their number of features. Note that the computed cardinalities are equal for all solvers. For *Linux* and *Automotive05*, the cardinality is unknown as no solver was able to compute a result. For the remaining systems, the cardinality ranges from $4.1 \cdot 10^9$ (*BerkeleyDB*) to $1.7 \cdot 10^{1534}$ (*Automotive02*).

Figure 8 shows the cardinality of the 13 successfully evaluated subject systems in relation to their number of features. There is a very weak positive correlation between the number of features and the cardinality (0.03 with Spearman). In several cases, a feature model with

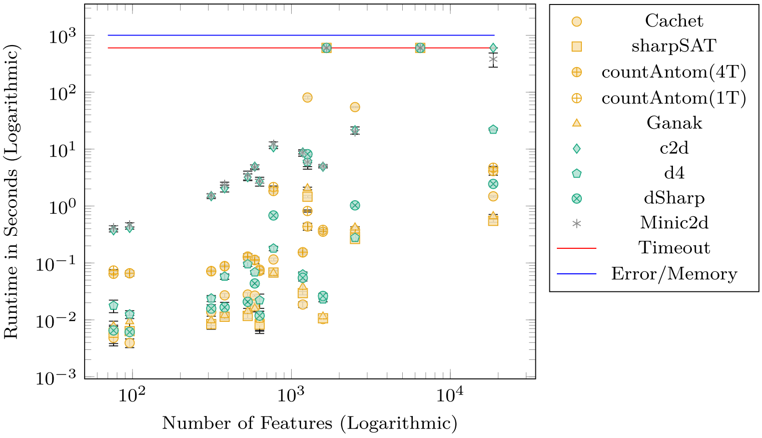

Fig. 6 Runtime of solvers in relation to the number of features

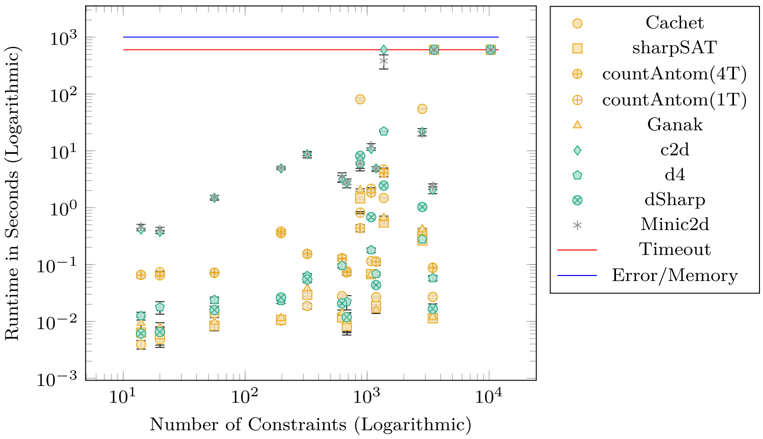

Fig. 7 Runtime of solvers in relation to the number of constraints

fewer features has a higher cardinality. For instance, *BusyBox* has 631 features and a cardinality of $2.1 \cdot 10^{201}$ while *FinancialServices* has 771 features and a cardinality of $9.7 \cdot 10^{13}$. Still, the three feature models with the largest number of features also have the highest cardinality. For instance, *Automotive02* has by far the largest cardinality $1.7 \cdot 10^{1534}$ and also seven times more features than *Automotive01* which has the second-highest number of features (disregarding Linux as we do not know its cardinality).

6.2 Experiment two: approximate #SAT solvers

Experiment 2a Figure 9 shows the runtimes of both evaluated approximate #SAT solvers on each feature model. `ApproxMC` hit the timeout of ten minutes for each but the two

Table 10 Cardinalities of Subject Systems (Sorted by #Features)

Subject systems	Number of valid configurations
BerkeleyDB	$4.1 \cdot 10^9$
axTLS	$8.3 \cdot 10^{11}$
uClibc	$1.7 \cdot 10^{40}$
uClinux-base	$2.6 \cdot 10^{22}$
Automotive04	$2.5 \cdot 10^{21}$
Automotive03	$2.5 \cdot 10^{31}$
BusyBox	$2.1 \cdot 10^{201}$
FinancialServices	$9.7 \cdot 10^{13}$
Embtoolkit	$5.1 \cdot 10^{96}$
CDL (116 Models)	$2.6 \cdot 10^{118} – 3.0 \cdot 10^{136}$
uClinux-distribution	$4.1 \cdot 10^{409}$
Automotive05	unknown
Automotive01	$5.4 \cdot 10^{217}$
Linux	unknown
Automotive02	$1.7 \cdot 10^{1534}$

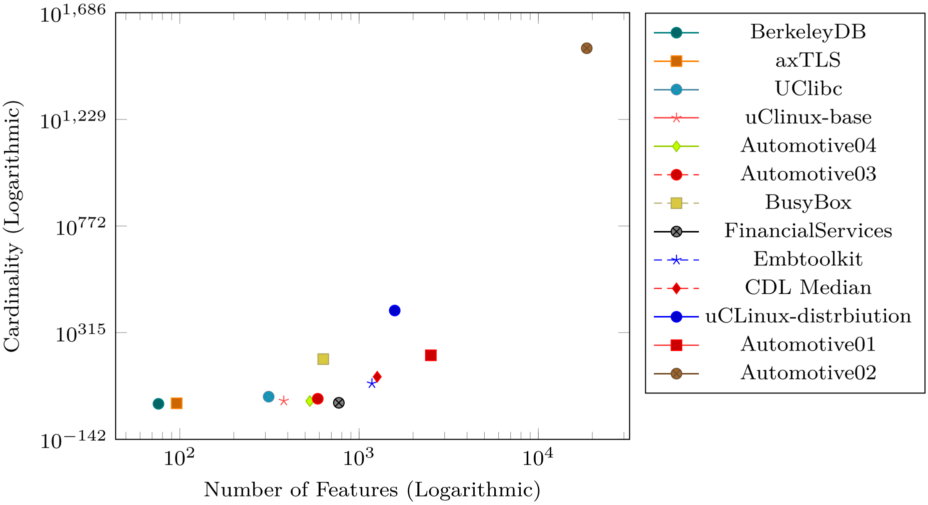

Fig. 8 Cardinality of subject systems in relation to number of features

smallest models, namely *axTLS* (96 features) and *BerkeleyDB* (76 features). Consequently, `ApproxMC` was excluded for *Experiment 2b*. `ApproxCount` hit the timeout for four subject systems.

Experiment 2b Figure 10 shows the runtimes of the best performing approximate #SAT solver (`ApproxCount`) with the exact #SAT solver that required the least time overall, namely `sharpSAT`. `ApproxCount` hit the timeout for four subject systems, while

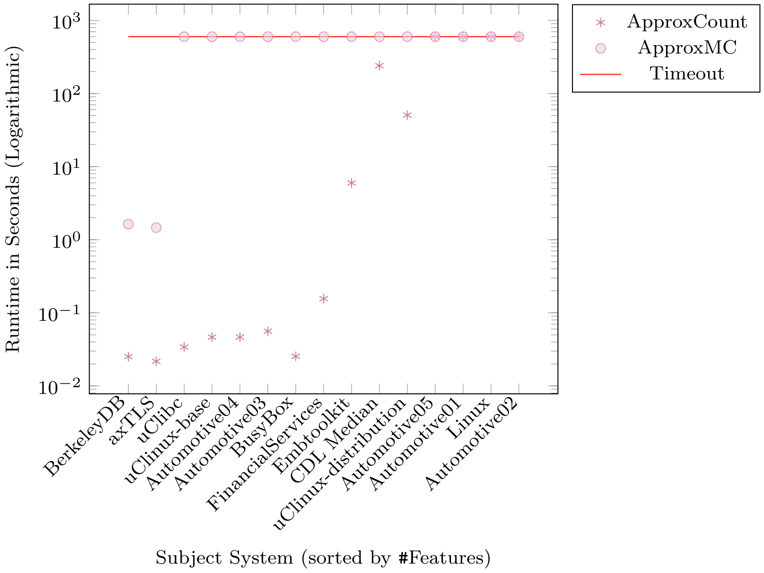

Fig. 9 Runtime in seconds for approximate #SAT solvers

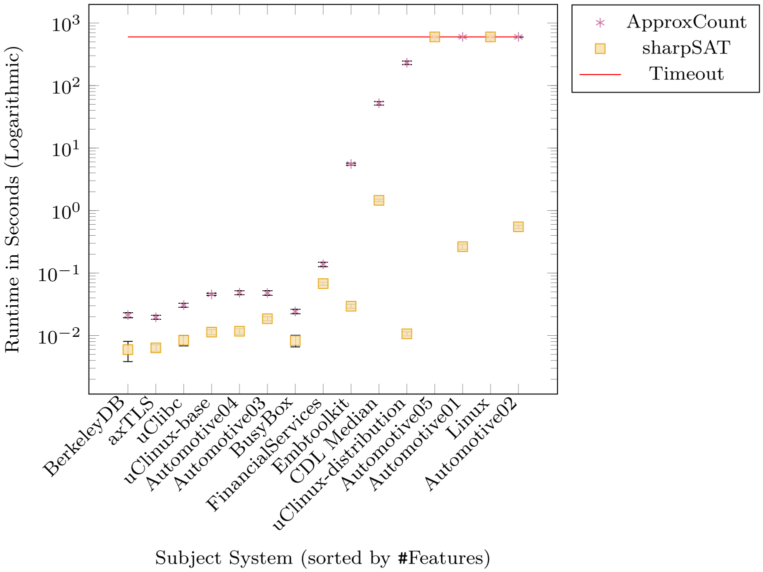

Fig. 10 Comparison runtime (median & standard deviation) ApproxCount vs sharpSAT

sharpSAT hit the timeout for two subject systems. For all 13 feature models that were successfully evaluated by at least one solver, sharpSAT is significantly faster than ApproxCount ($p < 0.0002$) with very large ($d > 5.1$) effect sizes for every single feature model. Furthermore, ApproxCount needs 5 minutes for 11 out of 15 systems while sharpSAT requires less than 2 seconds for those.

7 Discussion

In this section, we discuss the results regarding our research questions.

RQ1 *How do #SAT solvers perform on industrial feature models?* Our results indicate that the scalability of the #SAT solvers depends on the evaluated feature model. Based on our results, we expect that most industrial feature models can be evaluated within minutes or even seconds by the faster #SAT solvers we identified. Overall, 13 of the 15 analyzed feature models were successfully evaluated within 10 minutes. In addition, the fastest solver for each of those feature models required even less than one second which we consider scalable as it satisfies typical time restrictions of interactive environments and continuous integration environments. Nevertheless, there are systems for which no available #SAT solver scales. In our experiment, two systems, namely *Automotive05* and *Linux*, could not be evaluated by any solver not even within a timeout of 24 hours. Our results indicate that the hardness of both systems lies in their high number of features and constraints (c.f. the answer for **RQ3**).

Eight solvers, namely sharpSAT, Ganak, countAntom, dSharp, d4, Cachet, MiniC2D, and c2d, successfully evaluated 13 of 15 systems within 10 minutes of overall runtime. sharpSAT requires the least time to evaluate the 13 subject systems (2.6

seconds overall) closely followed by Ganak (3.4 seconds). While some solvers performed overall better than others, none of the solver is superior to the other solvers on every feature model. The results indicate that some solvers are inferior regarding the task of computing the cardinality of feature models, namely PicoSAT, Relsat, SharpCDCL, McTW, SUMC1, CNF2OBDD, BuDDy, Cudd, CNF2EADT, and SDD. Those solvers hit the timeout for at least four subject systems and some even for all systems while being substantially slower for the systems they successfully evaluated.

RQ2 *How do different classes of #SAT solvers perform on industrial feature models?* For single #SAT invocations, as performed in the experiment design at hand, we recommend the usage of the fastest DPLL-based solvers. The three best performing solvers, namely sharpSAT, Ganak, and countAntom are based on exhaustive DPLL.

For multiple #SAT invocations, reusing d-DNNFs seems promising. All d-DNNF compilers are part of the eight fastest solvers. For each feature model, that was successfully evaluated by at least one solver, the fastest d-DNNF-based solvers dSharp and d4 require at most a few seconds in sum for compilation and model counting. For each follow-up computation, the compiled d-DNNF could be re-used (e.g., for computing the number of valid configurations containing certain features). Hence, we expect d-DNNF solvers are likely faster when performing multiple computations, which is required for the majority of counting-based analyses (Sundermann et al. 2021). SDDs can also be reused and, thus, are a considerable candidate but the best performing SDD-based solver (MiniC2D) was substantially slower than dSharp (42 times slower) and d4 (18 times slower).

The remaining types of #SAT solvers, namely algebraic-based, BDDs, and other knowledge compilation formats performed substantially worse than the eight fastest solvers. Both algebraic solvers, namely ADDMC and SUMC1, overall successfully evaluated only nine (60%) of the subject systems. Hence, we excluded both solvers after Experiment 1a, and we cannot recommend using these solvers for #SAT-based analysis of feature models. The three BDD libraries overall successfully evaluated only six (40%) subject systems. BuDDy and Cudd even hit the timeout for 12 of 15 subject systems. Therefore, we do not recommend to use current BDD libraries for computing the cardinality of feature models. Nevertheless, BDDs are tractable (i.e., have polynomial time complexity w.r.t. to the size of the BDD) for additional computation types, such as existential quantification (Bryant 2018). Using BDDs for other feature-model analyses may thus still be beneficial.

RQ3 *How does the runtime of #SAT solvers correlate to structural metrics of the feature model?* The runtime required to compute the cardinality of a feature model generally increases if the feature model grows in size or complexity. There is a strong or very strong positive correlation between the runtime of #SAT solvers and several structural metrics related to size and complexity, namely number of features, number of leaf features, number of constraints, number of clauses, and number of literals.

Feature models with few features or constraints seem to be simple to analyze for #SAT solvers. Each subject system with less than 1,000 features was evaluated within one second by at least one solver, independent of the number of constraints. Analogously, all subject systems with less than 1,000 constraints were evaluated by at least one solver within one second, independent of the number of features.

While both systems for which no solver computed a result have at least 3,500 constraints, a large number of features, or constraints do not necessarily cause a time-consuming computation. The *Automotive02* system contains by far the most features

(18,616), but sharpSAT still evaluated it in less than a second. The reason probably lies in the comparatively low number of constraints (1,369) while *Linux* and *Automotive05* contain 3,545 and 10,321 constraints, respectively. Furthermore, *uClinux-base* contains 3,455 constraints but the fastest solver is about 50 times faster than for *Automotive02* which contains only 1,369 constraints.

Our insights indicate that, independent of structural metrics, one of the fastest solvers should be used. There is no single feature model for which one of the fastest eight solvers fails, while another #SAT solver computes a result. Thus, we expect that if the fastest solvers do not scale to a feature model, the others will also fail.

Predicting performance based on structural metrics may still be beneficial. For instance, countAntom is slower than sharpSAT for 12 out 13 (successfully evaluated) subject systems, but significantly faster for all 116 CDL feature models. Applying a meta #SAT solver that selects a suitable #SAT solver for a given feature model should yield runtime benefits. Our insights on the correlations between structural metrics and runtime may be a useful starting point for future work on predicting performance. In particular, the metrics showing a strong correlation are promising indicators for predicting performance.

RQ4 *How do approximate #SAT solvers perform on industrial feature models?* Approximating the results with the evaluated approximate #SAT solvers yields no benefits as we can acquire exact results with shorter runtimes. In particular, the fastest exact solver sharpSAT is significantly faster than both approximate #SAT solvers for every single successfully evaluated feature model. The slower solver ApproxMC computed a result only for the two smallest considered feature models. While ApproxCount computed a result for the majority of models, it scaled to fewer feature models than the fastest exact #SAT solver sharpSAT.

A reason for the worse performance of approximate #SAT solvers may be that the solvers were evaluated on (and eventually optimized for) formulas from different domains with generally fewer satisfying assignments. The largest formulas evaluated induce up to 10^{12} (Chakraborty et al. 2013) and up to 10^{33} (Wei and Selman 2005) satisfying assignments, respectively (compared to up to 10^{1534} in our evaluation). Optimizing those approximations for formulas representing feature models may be beneficial.

8 Threats to validity

We identified the following potential threats to validity for our evaluation.

Translating the subject systems to feature models It is possible that the translation from the original proprietary format into a feature model changes the variability. Knüppel et al. remarked some threats to internal validity regarding their translation of product lines (Knüppel et al. 2017). First, there are differences between feature model semantics and the semantics of the variability languages used for CDL and KConfig. Second, the translation may have removed a few cross-tree constraints. Third, a few cases lead to features that did not appear in the input format (Knüppel et al. 2017). Still, this is the largest available benchmark and has been used by other authors (Krieter et al. 2018; Plazar et al. 2019; Baranov et al. 2020). Pett et al. (2021) translated the BusyBox model to CNF using KClause (Oh et al. 2019).[19] Then, the authors translated the CNF into feature model that is equivalent to

[19] https://doi.org/10.5281/zenodo.2574218

the CNF and, thus, should maintain the variability. In addition to publicly available subject systems, we translated three automotive product lines into feature models from a proprietary format. It is possible that we misinterpreted given constraints. However, we created the parser in direct cooperation with company interns. Furthermore, the interns reviewed the resulting feature models.

Translating feature models to the DIMACS format An incorrect translation of feature models to CNF may lead to incorrect cardinalities. Another important aspect of the translation to CNF is that the number of satisfying assignments has to be equal for the resulting CNF. This is not given for every conversion method (Knüppel et al. 2017). For every translation to CNF in DIMACS format, we used the FeatureIDE library (Kästner et al. 2009). FeatureIDE uses a transformation that does not introduce new variables nor changes the number of solutions. Nevertheless, we performed the following sanity checks to ensure a correct translation. First, we manually computed the model count of small feature models (¡ 100 valid configurations and only few cross-tree constraints) and compared these results with the ones computed by the solvers. Second, we made changes to the feature model that should change the model count in a certain way. For example, we added an optional feature to the root which should always double the number of valid configurations and verified that the #SAT solvers computed the expected result.

We did not consider the time required to translate the feature model to CNF. However, the translation time is equivalent for all #SAT solvers as they use the same CNF. Furthermore, for all feature models but Linux the translation required only a few milliseconds.

Wrapper for BuDDy and Cudd As described in Section 5.2, we used a wrapper to interface with `BuDDy` and `Cudd`, due to their incapability to process DIMACS directly. The implementation of our wrapper for `BuDDy` and `Cudd` could be erroneous or inefficient, yielding a negative impact on their performance. While parsing of the input is handled by the wrapper, the BDDs are constructed entirely by `BuDDy` and `Cudd` using the parameters and techniques suggested in their respective manuals. For each successful computation of `BuDDy` and `Cudd`, the returned number of satisfying assignments was correct. Furthermore, for every feature model the parsing time of the wrapper required less than one second and at most 10% of the overall runtime. Note that each time `BuDDy` or `Cudd` required more than one second of runtime the relative share of parsing time is even lower (at most 2%). Thus, we consider the parsing time of the DIMACS input is negligible compared to the overall runtime and do not expect an impact on our conclusions on the performance of `BuDDy` or `Cudd`. Furthermore, we decided against using the widely used (Mendonça 2009; Mendonca and Cowan 2010; Pohl et al. 2011) library JavaBDD as it misses support for the latest version of `Cudd` (3.0.0) and frequently crashes when using `BuDDy`.

Parameterization of the solvers Typically, there are various parameters to adapt the behavior of the solvers, such as enabling or disabling boolean constraint propagation. These parameters may have a noticeable impact on the scalability in some cases (Wei and Selman 2005). In general, we used the default parameterization for each solver to achieve the following: (1) prevent introducing a bias based on our decision of the parameterization, and (2) evaluate the solver's performance when integrated without further expertise which we typically expect in practice. In general, evaluating multiple parameter permutations multiplicatively vastly increases the complexity, required time, and ecological footprint of the performed experiments.

Correctness of the solvers We used only external solvers without a possibility of directly verifying the results. However, for every subject system the number of satisfying assignments returned by each solver was equal. This is a strong indicator for the correctness of the solvers. Furthermore, we manually computed the cardinality for multiple small feature models (¡ 100 valid configurations) and compared them to the results of #SAT solvers.

Computational bias When performing measurements, it is possible that a program accelerates during the computations. In this case, early measurements might be slower than later ones. In our benchmark framework, each single invocation of a #SAT solver is performed in a separate execution of the binary. Thus, the solvers are in the same state at the start of each computation. It may be possible that hardware optimizations induce a warm-up. We disabled *turboboost* and *file system cache* to reduce a potential bias.

In general, it is possible for a background process to influence the runtime of a solver and, thus, impact our results. First, we disabled hyper-threading. Second, we performed a preliminary experiment with five repetitions for each solver several months prior to the evaluation described in this work which resulted in the same conclusions regarding the performance of the solvers. Third, during the 50 repetitions of Experiment 1b, the solvers always had very similar runtimes for the same feature model. Fourth, during the runtime of the experiments no other computational expensive task was performed on the device and each measurement was performed sequentially. Fifth, we occasionally monitored the available RAM and CPU resources. Every time we tracked, there were at least 40 GB of RAM available and less than 15% of the CPU used. Therefore, we do not expect that any conclusion we made is impacted by a background process.

Single measurements for slow solvers For slow solvers, we only performed one repetition per measurement. It is possible that for 50 repetitions the median significantly differs from a single measurement for some feature models. Nevertheless, neither solver was excluded after Experiment 1a due to single or few measurements but due to a large gap to the fastest #SAT solvers.

Random effects It is possible that the runtimes of #SAT solvers are affected by random effects. For example, `c2d` (Darwiche 2002, 2004) randomly chooses cuts in order to create a decomposition tree of the formula at the start of the computation. To reduce the bias resulting from randomness, we performed 50 repetitions in Experiment 1b and Experiment 2b and performed statistical tests on the significance of results.

External validity solvers Our results cannot be necessarily transferred to other #SAT solvers. For instance, Kübler etal (Kübler et al. 2010) developed their own tool to compute the cardinality of feature models. Their tool is not publicly available and, thus, we could not evaluate and compare it to other solvers. Nevertheless, we evaluated a large variety of different #SAT solvers. To the best of our knowledge, we included each publicly available #SAT solver in our benchmark.

External validity systems We cannot claim that our results can be transferred to any other industrial product lines. However, we considered multiple domains, namely automotive, operating system, database, and financial services to increase our confidence. We overall evaluated 130 feature models which cover a wide range of number of features (76–18,616), number of constraints (20–10,321), number of valid configurations ($\approx 10^9$–10^{1534}), and runtime of #SAT solvers (between few milliseconds and hitting a timeout of 24 hours).

Therefore, we expect that our results represent a reasonable indicator for the scalability of #SAT solvers on other product lines.

9 Related work

In this section, we discuss work that is related to ours regarding (1) applying #SAT to feature models, (2) usage of #SAT technology in feature-modelling tools, and (3) computing the cardinality with tools that are not based on propositional logic.

Applying #SAT to feature models Kübler et al. (2010) also evaluated the use of two #SAT solvers, `Cachet` (Sang et al. 2004) and `c2d` (Darwiche 2004), on three different versions of an automotive product line. We evaluated both solvers and they were outperformed by newer solvers on most instances. However, the authors also proposed their own model counter that was not based on conjunctive normal form and performed better than `Cachet` and `c2d`. However, their solver and their evaluated product lines are not publicly available. Therefore, we could not directly compare the results. Overall, we evaluated 21 solvers on 130 formulas while Kübler et al. evaluated 3 solvers on 3 formulas.

Pohl et al. (2011) evaluated different feature model analyses including model counting using BDDs, constraint-satisfaction-problem solvers, and SAT solvers. However, the authors used models with much smaller configuration spaces and fewer features for their evaluation. Their analyzed configuration spaces only reached up to 10^8 valid configurations whereas 97.3% of our feature models have larger configuration spaces with up-to 10^{1534} valid configurations.

Oh et al. (2019) evaluated the application of #SAT for uniform random sampling with their tool *Smarch*. Their results indicate that #SAT can be used to create a uniformly distributed sample for a variety of industrial feature models. However, their evaluation is limited to one application (uniform random sampling) and limited to one solver (`sharpSAT`). Sharma et al. (2018) proposed using #SAT technology for uniform random sampling and provided an algorithm exploiting d-DNNFs. However, their empirical evaluation is also limited to uniform random sampling and two solvers (`d4` and `dSharp`). We evaluate 21 solvers including the three solvers considered by Oh et al. (2019) and Sharma et al. (2018).

Current tool support for #SAT technology BDDs are a popular choice for counting the number of valid configurations in a product line as it is possible to compute the BDD offline and then compute the cardinality with linear time in the number of nodes (Acher et al. 2013; Hadzic et al. 2004; Mendonça et al. 2009). However, our results indicate that existing BDD libraries do not scale to industrial feature models. Additionally, d-DNNFs can be computed offline as well and performed significantly better than BDDs in all our experiments (Darwiche and Marquis 2002).

FeatureIDE uses a regular SAT solver (`SAT4J` (Le Berre and Parrain 2010)) to compute the number of valid configurations (Thüm et al. 2011). The tool realizes counting with a regular SAT solver using blocking clauses (Toda and Soh 2016); after finding a valid assignment α, the negation of α is added as a clause to the formula. Thus, α is not a valid assignment for the resulting formula and the next run of the solver returns another assignment until no new satisfying assignments are left. For each satisfying assignment (i.e., valid configuration), an invocation of the SAT solver is required. Our results indicate that industrial feature models induce up to 10^{1500} valid configurations. Therefore, the algorithm should not scale for larger systems.

Non-propositional model counting Constraint satisfaction problems (CSP) are an alternative to propositional logic for the representation of feature models (Benavides et al. 2005, 2006, 2010; Pohl et al. 2011). CSPs are defined by a set of variables, domains for each variable, and constraints over these variables. For CSPs, the variables may also be integers or intervals, contrary to propositional boolean variables which are strictly binary (Benavides et al. 2010). Benavides et al. (2005) use constraint programming (CP) to compute the number of valid configurations for feature models. However, the models considered in their experiment only included up to 23 features (Benavides et al. 2005). Pohl et al. (2011) compare SAT solvers, BDDs, and CSP solvers for several feature-model analyses that include computing the cardinality. Their results indicate that the analyzed CSP solvers scale far worse than the #SAT solvers evaluated in our experiment (Pohl et al. 2011). Munoz et al. (2019) examined counting the number of valid configurations of feature models with numerical features for uniform random sampling. The authors evaluated an SMT solver, a CP solver, and the #SAT solver `sharpSAT`. The numerical values were translated to propositional logic using bit-blasting (Munoz et al. 2019). In their experiment, `sharpSAT` outperformed the CP and SMT solver. This indicates that #SAT solvers are also a reasonable choice for computing the number of valid configurations for feature models with numerical values and our results (e.g., recommendations of solvers) could also be useful for non-propositional model counting.

10 Future work

In this section, we describe further tasks in applying #SAT solvers to industrial feature models.

Cardinality of features and partial configurations In this work, we limited our empirical evaluation to computing the cardinality of feature models (i.e., the number of valid configurations of the entire feature model). In our previous work (Sundermann et al. 2021), we presented 21 applications and a major part of them is dependent on the cardinality of (possibly many) features (i.e., number of valid configurations that contain a specific feature) or the cardinality of partial configurations (i.e., number of valid configurations that include some and exclude some other features). The runtimes of computing the cardinality of the entire feature model (as measured in our empirical evaluation) can be used as estimate for computing the cardinality of a feature or a partial configuration due to the similar input formulas (Sundermann et al. 2021). Nevertheless, to provide accurate insights on the scalability of these applications, an empirical evaluation for computing the cardinality of features and partial configurations is required.

Analyzing #SAT during the evolution of systems Often, product lines evolve over time (Svahnberg and Bosch 1999). Typically, underlying feature models grow both in number of features and constraints (Sundermann et al. 2020; Israeli and Feitelson 2010). As we found a strong correlation between the scalability of #SAT and both metrics (i.e., number of features and number of constraints), the evolution of a system may increase the runtime required to evaluate an underlying feature model with a #SAT solver. This is also indicated by the preliminary results of our previous work (Sundermann et al. 2020). If a product lines evolves over time, even product lines for which #SAT solvers scale currently may be infeasible to analyze in the future or vice versa.

Exploit d-DNNFs for cardinality-based analyses In our empirical evaluation, all three d-DNNF compilers, namely `dSharp`, `d4`, and `c2d` were part of the eight fastest solvers. If we require multiple computations on a single feature model (e.g., to compute the cardinalities for multiple features or partial configurations), exploiting a compiled d-DNNF may be beneficial. However, the research on exploiting an existing d-DNNF is very limited (Sharma et al. 2018) as most work focuses on the compilation process (Darwiche 2002, 2004; Lagniez and Marquis 2017; Oztok and Darwiche 2014; Muise et al. 2010; Huang and Darwiche 2005). While SDDs and BDDs are also considerable target formats for knowledge compilation, all compilers based on these formats performed significantly worse than `dSharp` and `d4`.

Parameterize #SAT solvers In this paper, we invoked the #SAT solvers using the default parameters with a few exceptions (e.g., some solvers require specific parameters to perform #SAT instead of SAT). Other parameterizations (e.g., selecting strategies for variable ordering) may improve the performance of #SAT solvers. Especially, the runtime of approximate #SAT solvers is dependent on the given parameters. However, identifying effective parameters is not trivial. To use #SAT solvers to their full potential requires finding suitable parameters that result in efficient and effective computations.

Further metrics for a meta-solver Our results show that the solvers perform differently depending on the system. None of the solvers is faster than all other solvers for every feature model. Analyzing structural metrics of the feature model may enable an efficient meta-solver that selects the most promising solver depending on a given instance. For regular SAT, it is already known that selecting a solver based on a given formula often improves the performance (Xu et al. 2008).

Directly translate feature models to target format For every experiment, we used propositional formulas in conjunctive normal form. The translation to CNF was not considered in the runtime. However, for the larger systems, the translation requires a considerable amount of time. Directly translating the feature model to knowledge compilation target formats, such as BDDs or d-DNNFs, might result in two benefits. First, the time overhead of translating the model to CNF would be eliminated. Second, using structural information of the feature model may accelerate the translation to the target format.

Purpose-built solvers for analyzing feature models None of the analyzed #SAT solvers and knowledge compilers is optimized for feature models. Optimizing the computations specifically for feature models may improve the performance of solvers. One improvement may be deriving beneficial variable orders using structural information of the feature model. The performance of each considered type of solver is highly dependent on variable ordering

(Sang et al. 2005; Muise et al. 2010; Thurley 2006; Wei and Selman 2005; Darwiche 2002; Toda and Soh 2016).

11 Conclusion

A large variety of feature-model analyses is dependent on computing the cardinality of features models (Sundermann et al. 2021). However, the scalability of such analyses is largely unknown. We analyzed 19 exact and 2 approximate #SAT solvers on the task of computing the cardinality of industrial feature models. Overall, we evaluated the #SAT solvers on 130 feature models from 15 subject systems.

Our results strongly indicate that current #SAT solvers scale to many, but not to all systems. Out of the 15 evaluated systems, eight solvers computed the cardinality of 13 (86.7%) systems within 10 minutes per system. The solver with the overall shortest runtime is `sharpSAT` requiring less than three seconds for all 13 models in total. However, for the two remaining systems, namely *Linux* and *Automotive05*, none of the solvers was able to compute a result within 24 hours of runtime.

While no solver was strictly superior to all other solvers, we identified several promising #SAT solvers for the task of computing the cardinality of feature models. For single #SAT computations on feature models, we recommend using the DPLL-based solvers `sharpSAT`, `countAntom`, and `Ganak`. For applications requiring multiple #SAT invocations, reusing d-DNNFs seems promising. All three considered d-DNNF compilers, namely `dSharp`, `d4`, and `c2d`, were within the fastest eight solvers. Surprisingly, each approximate #SAT solver we evaluated is significantly slower than the fastest exact #SAT solver for every considered feature model and, thus, yields no benefits over the exact solvers.

The runtime of all #SAT solvers tends to increase for feature models with a larger number of constraints or features. Each feature model with either fewer than 1,000 features or fewer than 1,000 constraints was evaluated within one second by the solver with the shortest runtime for that feature model. Nevertheless, the results indicate that a higher number of constraints or features does not necessarily result in longer runtimes.

Acknowledgments We sincerely thank the reviewers and editors for their valuable feedback on our work. Further, we thank Heiko Raab and Rahel Arens for their help in improving the manuscript. This work has been partially supported by the German Research Foundation within the project VariantSync (TH 2387/1-1).

Funding Open Access funding enabled and organized by Projekt DEAL.

Data Availability The datasets generated during and/or analyzed during the current study are available in the replication repository, https://doi.org/10.5281/zenodo.7329979.

Declarations

Conflict of Interests The authors have no conflicts of interest to declare that are relevant to the content of this article.

References

Acher M, Collet P, Lahire P, France RB (2013) Familiar: a domain-specific language for large scale management of feature models. Sci Comput Program (SCP) 78(6):657–681

Ananieva S, Kowal M, Thüm T, Schaefer I (2016) Implicit constraints in partial feature models. In: Proc. int'l workshop on feature-oriented software development (FOSD), ACM, pp 18–27

Ansótegui C, Bonet ML, Levy J (2009) On the structure of industrial SAT instances. In: Proc. int'l conf.on principles and practice of constraint programming (CP), Springer, pp 127–141

Apel S, Batory D, Kästner C, Saake G (2013) Feature-Oriented Software Product Lines. Springer

Artusi R, Verderio P, Marubini E (2002) Bravais-Pearson and spearman correlation coefficients: meaning, test of hypothesis and confidence interval. Int J Biol Markers (IJBM) 17(2):148–151

Bagheri E, Gasevic D (2011) Assessing the maintainability of software product line feature models using structural metrics. Softw Qual J (SQJ) 19(3):579–612

Bagheri E, Noia TD, Gasevic D, Ragone A (2012) Formalizing interactive staged feature model configuration. J Softw Evol Process 24(4):375–400

Baranov E, Legay A, Meel KS (2020) Baital: an adaptive weighted sampling approach for improved t-wise coverage. In: Proc. Europ. Software engineering conf./foundations of software engineering (ESEC/FSE), ACM, pp 1114–1126

Batory D (2005) Feature models, grammars, and propositional formulas. In: Proc. int'l systems and software product line conf. (SPLC), Springer, pp 7–20

Bayardo Jr RJ, Pehoushek JD (2000) Counting models using connected components. In: Proc. Conf. on artificial intelligence (AAAI), AAAI press, pp 157–162

Benavides D, Segura S, Ruiz-Cortés A (2010) Automated analysis of feature models 20 years later a literature review. Inf Syst 35(6):615–708

Benavides D, Segura S, Trinidad P, Ruiz-Cortés A (2006) Using java CSP solvers in the automated analyses of feature models. In: Proc. generative and transformational techniques in software engineering, Springer, pp 399–408

Benavides D, Segura S, Trinidad P, Ruiz-Cortés A (2007) FAMA: Tooling a framework for the automated analysis of feature models. In: Proc. int'l workshop on variability modelling of software-intensive systems (VaMoS), lero, pp 129–134. Technical report 2007-01

Benavides D, Trinidad P, Ruiz-Cortés A (2005) Using constraint programming to reason on feature models. In: Proc. Int'l conf. on software engineering and knowledge engineering (SEKE), pp 677–682

Bezerra CIM, Andrade RMC, Monteiro JMS (2014) Measures for quality evaluation of feature models. In: Proc. Int'l conf. on software reuse (ICSR), Springer, pp 282–297

Biere A (2008) PicoSAT Essentials. J. Satisfiability Boolean Model Comput 4:75–97

Biere A, Heule M, van Maaren H, Walsh T (2009) Handbook of satisfiability: volume 185 frontiers in artificial intelligence and applications. IOS press

Bosch J, Florijn G, Greefhorst D, Kuusela J, Obbink JH, Pohl K (2001) Variability issues in software product lines. In: Proc. int'l workshop on software product-family engineering (PFE), Springer, pp 13–21

Bryant RE (1986) Graph-Based Algorithms for boolean function manipulation. IEEE Trans Comput C-35(8):677–691

Bryant RE (2018) Binary decision diagrams. In: Handbook of model checking, Springer, pp 191–217

Burchard J, Schubert T, Becker B (2015) Laissez-Faire caching for parallel# SAT solving. In: Proc. Int'l conf. on theory and applications of satisfiability testing (SAT), Springer, pp 46–61

Chakraborty S, Meel KS, Vardi MY (2013) A scalable approximate model counter. In: Schulte C (ed) Proc. Int'l Conf. on Principles and Practice of Constraint Programming (CP), Springer, pp 200–216

Chen S, Erwig M (2011) Optimizing the product derivation process. In: Proc. Int'l systems and software product line conf. (SPLC), IEEE, pp 35–44

Conover WJ, Iman RL (1981) Rank transformations as a bridge between parametric and nonparametric statistics. Am Stat 35(3):124–129

Czarnecki K, Wàsowski A (2007) Feature diagrams and logics: There and back again. In: Proc. Int'l systems and software product line conf. (SPLC), IEEE, pp 23–34

Darwiche A (2001) On the tractable counting of theory models and its application to truth maintenance and belief revision. J Appl Non-Class Logics 11(1-2):11–34

Darwiche A (2002) A compiler for deterministic, decomposable negation normal form. In: Proc. Conf. on artificial intelligence (AAAI), AAAI press, pp 627–634

Darwiche A (2004) New advances in compiling CNF to decomposable negation normal form. In: Proc. Europ. Conf. on artificial intelligence, IOS press, pp 318–322

Darwiche A (2011) SDD: a new canonical representation of propositional knowledge bases. AAAI Press, pp 819–826

Darwiche A, Marquis P (2002) A knowledge compilation map. J Artif Intell Res (JAIR) 17(1):229–264

Fernández-Amorós D, Heradio R, Cerrada JA, Cerrada C (2014) A scalable approach to exact model and commonality counting for extended feature models. IEEE Trans Softw Eng (TSE) 40(9):895–910

Fichte JK, Hecher M, Hamiti F (2021) The model counting competition 2020. ACM J Exp Algorithmics (JEA) 26:1–26 , 13:1–26

Fichte JK, Hecher M, Zisser M (2019) An Improved GPU-based SAT Model Counter. In: Proc. Int'l conf. on principles and practice of constraint programming (CP), Springer, pp 491–509

Fritsch C, Abt R, Renz B (2020) The benefits of a feature model in banking. In: Proc. Int'l systems and software product line conf. (SPLC). ACM

Galindo JA, Acher M, Tirado JM, Vidal C, Baudry B, Benavides D (2016) Exploiting the enumeration of all feature model configurations a new perspective with distributed computing. In: Proc. Int'l systems and software product line conf. (SPLC), ACM, pp 74–78

Gomes CP, Sabharwal A, Selman B (2006) Model counting: a new strategy for obtaining good bounds. In: Proc. Conf. on artificial intelligence (AAAI), AAAI press, pp 54–61

Hadzic T, Subbarayan S, Jensen RM, Andersen HR, Møller J, Hulgaard H (2004) Fast backtrack-free product configuration using a precompiled solution space representation. In: Proc. Int'l conf. on economic, technical and organisational aspects of product configuration systems, Gamez publishing, pp 131–138

Heradio R, Fernández-Amorós D, Cerrada JA, Abad I (2013) A literature review on feature diagram product counting and its usage in software product line economic models. Int J Softw Eng Knowl Eng (IJSEKE) 23(08):1177–1204

Heradio R, Fernández-Amorós D, Mayr-Dorn C, Egyed A (2019) Supporting the statistical analysis of variability models. In: Proc. Int'l conf. on software engineering (ICSE), IEEE, pp 843–853

Heradio R, Pérez-Morago HJ, Fernández-Amorós D, Bean R, Cabrerizo FJ, Cerrada C, Herrera-Viedma E (2016) Binary decision diagram algorithms to perform hard analysis operations on variability models. In: Proc. Int'l Conf. on Intelligent Software Methodologies, Tools and Techniques (SOMET), IOS Press, pp 139–154

Heradio-Gil R, Fernández-Amorós D, Cerrada JA, Cerrada C (2011) Supporting commonality-based analysis of software product lines. IET Softw 5(6):496–509

Heß T, Sundermann C, Thüm T (2021) On the scalability of building binary decision diagrams for current feature models. In: Proc. Int'l systems and software product line conf. (SPLC), ACM, pp 131–135

Huang J, Darwiche A (2005) DPLL with a trace: from SAT to knowledge compilation. In: Proc. Int'l joint conf. on artificial intelligence (IJCAI), vol 5. Professional book center, pp 156–162

Huth M, Ryan M (2004) Logic in computer science: modelling and reasoning about systems. Cambridge University Press

Israeli A, Feitelson DG (2010) The linux kernel as a case study in software evolution. J. Syst Softw (JSS) 83(3):485–501

Kästner C, Apel S, Batory D (2007) A case study implementing features using aspectJ. In: Proc. Int'l systems and software product line conf. (SPLC), IEEE, pp 223–232

Kästner C, Thüm T, Saake G, Feigenspan J, Leich T, Wielgorz F (2009) FeatureIDE: a tool framework for feature-oriented software development. In: Proc. Int'l Conf. on Software Engineering (ICSE), IEEE, pp 611–614. Formal demonstration paper

Klebanov V, Manthey N, Muise C (2013) SAT-based analysis and quantification of information flow in programs. In: Proc. Int'l conf. on quantitative evaluation of systems (QEST), Springer, pp 177–192

Knüppel A, Thüm T, Mennicke S, Meinicke J, Schaefer I (2017) Is there a mismatch between Real-World feature models and Product-Line research? In: Proc. Europ. software engineering conf. Foundations of software engineering (ESEC/FSE), ACM, pp 291–302

Koriche F, Lagniez J-M, Marquis P, Thomas S (2013) Knowledge compilation for model counting affine decision trees. In: Proc. Int'l joint conf. on artificial intelligence (IJCAI), AAAI press

Kowal M, Ananieva S, Thüm T (2016) Explaining anomalies in feature models. In: Proc. Int'l conf. on generative programming: concepts & experiences (GPCE), ACM, pp 132–143

Krieter S, Pinnecke M, Krüger J, Sprey J, Sontag C, Thüm T, Leich T, Saake G (2017) Feature IDE empowering third-party developers. In: Proc. Int'l systems and software product line conf. (SPLC), ACM, pp 42–45

Krieter S, Thüm T, Schulze S, Schröter R, Saake G (2018) Propagating configuration decisions with modal implication graphs. In: Proc. Int'l conf. on software engineering (ICSE), ACM, pp 898–909

Kübler A, Zengler C, Küchlin W (2010) Model counting in product configuration. In: Proc. Int'l workshop on logics for component configuration (lococo), Open publishing association, pp 44–53

Lagniez J-M, Marquis P (2017) An improved decision-DNNF compiler. In: Proc. Int'l joint conf. on artificial intelligence (IJCAI), pages 667–673. International joint conferences on artificial intelligence

Lagniez J-M, Marquis P, Szczepanski N (2018) DMC: a distributed model counter. In: Proc. Int'l joint conf. on artificial intelligence (IJCAI), pp 1331–1338. International joint conferences on artificial intelligence

Le Berre D, Parrain A (2010) The sat4j library, release 2.2. J Satisfiability Boolean Model Comput 7(2-3):59–64

McKnight PE, Najab J (2010) Mann-Whitney U test. Corsini Encycl Psychol 1–1

Mendonca M, Cowan D (2010) Decision-making coordination and efficient reasoning techniques for Feature-Based configuration. Sci Comput Program (SCP) 75(5):311–332

Mendonça M (2009) Efficient reasoning techniques for large scale feature models. University of Waterloo, PhD thesis

Mendonça M, Branco M, Cowan D (2009) S.P.L.O.T.: software product lines online tools. In: Proc. Conf. on object-oriented programming, systems, languages and applications (OOPSLA), ACM, pp 761–762

Mendonça M, Wàsowski A, Czarnecki K (2009) SAT-Based analysis of feature models is easy. In: Proc. Int'l systems and software product line conf. (SPLC), Software engineering institute, pp 231–240

Muise C, McIlraith S, Beck JC, Hsu E (2010) Fast d-DNNF Compilation with sharpSAT. In: Proc. Conf. on artificial intelligence (AAAI). AAAI press

Munoz D-J, Oh J, Pinto M, Fuentes L, Batory D (2019) Uniform random sampling product configurations of feature models that have numerical features. In: Proc. Int'l systems and software product line conf. (SPLC), ACM, pp 289–301

Nieke M, Mauro J, Seidl C, Thüm T, Yu IC, Franzke F (2018) Anomaly analyses for Feature-Model evolution. In: Proc. Int'l conf. on generative programming: Concepts & experiences (GPCE), ACM, pp 188–201

Oh J, Batory D, Myers M, Siegmund N (2016) Sampling: finding product line configurations with high performance by random technical report University of Texas at Austin

Oh J, Batory D, Myers M, Siegmund N (2017) Finding Near-Optimal configurations in product lines by random sampling. In: Proc. Int'l symposium on foundations of software engineering (FSE), pp 61–71

Oh J, Gazzillo P, Batory D, Heule M, Myers M (2019) Uniform Sampling from Kconfig Feature Models. Technical Report TR-19-02, The university of texas at austin department of computer science

Oztok U, Darwiche A (2014) On compiling CNF into decision-DNNF. In: Proc. Int'l conf. on principles and practice of constraint programming (CP), Springer, pp 42–57

Oztok U, Darwiche A (2015) A Top-Down compiler for sentential decision diagrams. In: Proc. Int'l joint conf. on artificial intelligence (IJCAI), AAAI press, pp 3141–3148

Pérez Morago HJ (2016) BDD Algorithms to perform hard analysis operations on variability models. Universidad Nacional de Educación a Distancia, PhD thesis

Perrouin G, Sen S, Klein J, Baudry B, Le Traon Y (2010) Automated and scalable T-Wise test case generation strategies for software product lines. In: Proc. Int'l conf. on software testing, verification and validation (ICST), IEEE, pp 459–468

Pett T, Krieter S, Runge T, Thüm T, Lochau M, Schaefer I (2021) Stability of Product-Line sampling in continuous integration. In: Proc. Int'l working conf. on variability modelling of software-intensive systems (VaMoS). ACM

Pett T, Thüm T, Runge T, Krieter S, Lochau M, Schaefer I (2019) Product sampling for product lines the scalability challenge. In: Proc. Int'l systems and software product line conf. (SPLC), ACM, pp 78–83

Plazar Q, Acher M, Perrouin G, Devroey X, Cordy M (2019) Uniform sampling of SAT solutions for configurable systems are we there yet? In: Proc. Int'l conf. on software testing, verification and validation (ICST), IEEE, pp 240–251

Pohl R, Lauenroth K, Pohl K (2011) A performance comparison of contemporary algorithmic approaches for automated analysis operations on feature models. In: Proc. Int'l conf. on automated software engineering (ASE), IEEE, pp 313–322

Rudell R (1993) Dynamic variable ordering for ordered binary decision diagrams. In: Proc. Int'l conf. on computer-aided design (ICCAD), IEEE, pp 42–47

Sang T, Bacchus F, Beame P, Kautz HA, Pitassi T (2004) Combining component caching and clause learning for effective model counting. In: Proc. Int'l conf. on theory and applications of satisfiability testing (SAT), Springer, pp 20–28

Sang T, Beame P, Kautz H (2005) Heuristics for fast exact model counting. In: Proc. Int'l conf. on theory and applications of satisfiability testing (SAT), Springer, pp 226–240

Schröter R, Krieter S, Thüm T, Benduhn F, Saake G (2016) Feature-model interfaces: the highway to compositional analyses of Highly-Configurable systems. In: Proc. Int'l conf. on software engineering (ICSE), ACM, pp 667–678

Segura S (2008) Automated analysis of feature models using atomic sets. In: Proc. Int'l systems and software product line conf. (SPLC), vol 2. IEEE, pp 201–207

Sharma S, Gupta R, Roy S, Meel KS (2018) Knowledge compilation meets uniform sampling. In: Proc. Int'l conf. on logic for programming, artificial intelligence, and reasoning, Easy chair, pp 620–636

Sharma S, Roy S, Soos M, Meel KS (2019) GANAK: a scalable probabilistic exact model counter. In: Proc. Int'l joint conf. on artificial intelligence (IJCAI), vol 19. AAAI press, pp 1169–1176

Sobernig S, Apel S, Kolesnikov S, Siegmund N (2016) Quantifying structural attributes of system decompositions in 28 Feature-Oriented software product lines. Empir Softw Eng (EMSE) 21(4):1670–1705

Sprey J, Sundermann C, Krieter S, Nieke M, Mauro J, Thüm T, Schaefer I (2020) SMT-based variability analyses in featureIDE. In: Proc. Int'l working conf. on variability modelling of software-intensive systems (VaMoS). ACM

Sullivan GM, Feinn R (2012) Using effect size—or why the P value is not enough. J Grad Med Educ 4(3):279–282

Sundermann C, Nieke M, Bittner PM, Heß T, Thüm T, Schaefer I (2021) Applications of #SAT solvers on feature models. In: Proc. Int'l working conf. on variability modelling of software-intensive systems (VaMoS). ACM

Sundermann C, Thüm T, Schaefer I (2020) Evaluating #SAT solvers on industrial feature models. In: Proc. Int'l working conf. on variability modelling of software-intensive systems (VaMoS). ACM

Svahnberg M, Bosch J (1999) Evolution in software product lines two cases. J Softw Maint (JSM) 11(6):391–422

Thüm T, Kästner C, Erdweg S, Siegmund N (2011) Abstract features in feature modeling. In: Proc. Int'l systems and software product line conf. (SPLC), IEEE, pp 191–200

Thurley M (2006) SharpSAT - counting models with advanced component caching and implicit BCP. In: Proc. Int'l conf. on theory and applications of satisfiability testing (SAT), Springer pp 424–429

Toda T, Soh T (2016) Implementing efficient all solutions SAT solvers. ACM J Exp Algorithmics (JEA) 21(1):1.12:1–1.12:44

Valiant LG (1979) The complexity of computing the permanent. Theor Comput Sci 8(2):189–201

Wei W, Selman B (2005) A new approach to model counting. In: Proc. Int'l conf. on theory and applications of satisfiability testing (SAT), Springer, pp 324–339

Xu L, Hutter F, Hoos HH, Leyton-Brown K (2008) SATzilla: portfolio-based algorithm selection for SAT. J Artif Intell Res (JAIR) 32:565–606

Zar JH (1972) Significance testing of the spearman rank correlation coefficient. J Am Stat Assoc 67(339):578–580

Affiliations

Chico Sundermann[1] · Tobias Heß[1] · Michael Nieke[2] · Paul Maximilian Bittner[1] · Jeffrey M. Young[3] · Thomas Thüm[1] · Ina Schaefer[2]

Tobias Heß
tobias.hess@uni-ulm.de

Michael Nieke
m.nieke@tu-bs.de

Paul Maximilian Bittner
paul.bittner@uni-ulm.de

Jeffrey M. Young
jeffrey.young@iohk.io

Thomas Thüm
thomas.thuem@uni-ulm.de

Ina Schaefer
i.schaefer@tu-bs.de

[1] University of Ulm, Ulm, Germany

[2] Technische Universität Braunschweig, Brunswick, Germany

[3] IOHK, Longmont, Colorado, United States