Evaluating #SAT Solvers on Industrial Feature Models

Chico Sundermann
Technische Universität Braunschweig

Thomas Thüm
University of Ulm

Ina Schaefer
Technische Universität Braunschweig

ABSTRACT

Configurable systems are widely used for families of products that share multiple configuration options. These systems often induce a large configuration space. Handling the variability of such a system is difficult without being able to measure its complexity. Several methods depend on computing the number of valid configurations, such as estimating the effort of an update or effectively reducing the variability of a system. In many cases, it is possible to map a configurable system to propositional logic. Therefore, we use #SAT in order to evaluate variability of such systems. A #SAT solver computes the number of valid assignments of a propositional formula. However, this problem is even harder than SAT. The main contribution of our work is an investigation of the scalability of off-the-shelf #SAT solvers on industrial feature models. Additionally, we examine the correlation between size of a system and the runtime of a solver computing the number of valid configurations. In this paper, we empirically evaluate nine publicly available #SAT solvers on 127 industrial feature models. Our results indicate that current solvers master a majority of the evaluated systems. However, there are large models, for which none of the evaluated solvers scales. Nevertheless, there are even larger and more complex systems for which the solvers scale.

KEYWORDS

Configurable Systems, Feature Models, Product Lines, Model Counting, Configuration Counting, #SAT

1 INTRODUCTION

A configurable system represents a variety of valid configurations that share certain configuration options, also called features [7]. A configuration is induced by a selection of the features. However, systems typically contain constraints which limit the set of valid configurations. For example, a car product line can be interpreted as such a configurable system. A constraint to limit the set of valid configurations might be allowing only one gearbox type for each car: manual or automatic. Thus, a configuration with both or no type of gearbox is considered invalid.

The set of valid configurations is known as the configuration space [5]. When introducing a new feature to a configurable, system the number of valid configurations is doubled in the worst case. In this scenario, each previously valid configuration is still valid with and without the new feature. Thus, the configuration space grows up-to exponentially. A configurable system with n features has up-to 2^n valid configurations [9]. However, real-world systems are typically more limited [8, 27].

It is difficult to manage this variability without knowing the number of valid configurations induced by a configurable system. For instance, the following methods rely on counting the number of valid configurations: commonality and homogenity of features [9, 14, 17], variability reduction [9], rating errors in a configurable system [27], and uniform random sampling [33]. These use cases are further described in Section 2.

Model counting can be used for quantitative analysis of configurable systems [17, 27]. In this paper, we focus on propositional model counting (for short #SAT), which determines the number of valid assignments for a given propositional formula. #SAT is an even harder problem than SAT [12]. However, the #SAT community keeps pushing the limits of #SAT solvers [6, 12, 15, 28, 42]. While Mendonca et al. argued that SAT is typically easy for real-world models compared to hard instances of SAT [30], this is currently unknown for #SAT to the best of our knowledge.

The main goal of this paper is to examine the efficiency of current #SAT solvers on real-world configurable systems. In order to do so, we analyze nine publicly available solvers [6, 10, 12, 15, 28, 32, 39, 42, 43] on 127 industrial systems. Hereby, we focus on exact #SAT solvers, contrary to approximate solvers. The analyzed systems consist of a benchmark from Knüppel et al. [25] and three automotive product lines provided by our industry partner. In our evaluation, we aim to answer the following research questions:

- **RQ1:** Do #SAT solvers scale to industrial configuration spaces?
- **RQ2:** Is one #SAT solver superior to the other solvers?
- **RQ3:** Does the runtime of the solvers correlate to the size or complexity of the configuration space?
- **RQ4:** How does the number of valid configurations relate to the number of all configurations?
- **RQ5:** How does the number of valid configurations change during the evolution of a configurable system?

2 THE NEED FOR CONFIGURATION COUNTING

In this section, we describe use cases for counting the number of valid configurations of configurable systems.

Commonality. The relative share of valid configurations that contain a certain feature is called commonality of that feature [17]. It can be used as an indication for the feature's relevancy in the configurable system [9, 17]. However, to compute the commonality, it is necessary to be able to count the number of valid configurations.

Homogeneity. The metric describes the similarity of valid configurations induced by the configurable system. If the common base of features in different configurations is small, this might indicate that using a product line instead of single products might not pay

VaMoS '20, February 5–7, 2020, Magdeburg, Germany

ACM ISBN 978-1-4503-7501-6/20/02…$15.00
https://doi.org/10.1145/3377024.3377025

off. A common way to compute homogeneity is the commonality mean over all features [14, 17].

Rating Errors. Counting the number of products can help to rate the impact of an error. After identifying an erroneous subset of the system, it is possible to compute the number of products containing this subset. An error appearing in more products might indicate a more critical error [27].

Variability Reduction. One might try to decrease the size of the configuration space by introducing further limiting constraints. However, in order to grasp the impact of such changes it is necessary to know the number of products before and after adding a new constraint [9].

Uniform Random Sampling. As it is mostly not feasible to analyze a configuration space by enumerating all configurations, it is common to create representative samples for a configurable system [33]. However, finding these samples is not easy. Randomly selecting features often results in invalid configurations. Only including the resulting valid configurations still does not guarantee randomness [35]. One way to achieve random samples is uniform random sampling [33]. The goal is to create a bijection between integers and configurations. Then, by randomly selecting an integer within the range, each configuration has the same probability to be included in the sample. The bijection can be achieved using #SAT by recursively assigning the variables [33]. For each assignment, the number of valid configurations needs to be computed [36]. This requires an efficient model counter, especially for large systems [33].

3 MOTIVATING EXAMPLE

Figure 1 shows a feature diagram representing a simplified car product line. It displays the tree structure and additional cross-tree constraints given in propositional logic. The tree structure and the cross-tree constraints limit the set of valid configurations. Each car of the product line is required to have a *Carbody*. This is indicated by the mandatory property of the feature. In contrast, a *Radio* is an optional feature. A configuration that does not contain exactly one of the *Gearbox* types, *Manual* or *Automatic*, is invalid, as they appear in an alternative-relation in the feature diagram. Furthermore, the *Ports* of a *Radio* include at least one of *USB* and *CD*. This relation is described by an or-relation. The cross-tree constraint *Navigation* $\Rightarrow$ *USB* represents that a car with *Navigation* requires an *USB*-port.

Feature diagrams are a visual representation of feature models [7]. They are commonly used to define configurable systems [7]. Each configuration corresponds to a selection of the features contained in the model. Knowing the exact number of configurations can help analyzing the feature model. For example, the feature *USB* is in 32 of the 42 (76,1%) valid configurations, while *CD* is only in 20 (47,6%). This relative share is called commonality and indicates the relevancy of a feature. During the development of the displayed car product line, it might be more effective to prioritize *Carbody* over *Radio* to build as many cars as possible with few features. Another scenario might be aiming to reduce the number of different cars induced by the product line for better maintainability. Suppose, the developers consider one of three options to reduce the number of configurations: removing *Bluetooth*, allowing only one of *USB* and *CD*, or making *Radio* a mandatory feature. Computing the number of configurations after each of the changes shows the variability reduction of each change. The resulting feature models contain 22, 26, and 40 valid configurations respectively. Therefore, removing *Bluetooth* is the most effective change regarding the reduction of variability. To enable such analyses, we are interested in computing the number of valid configurations.

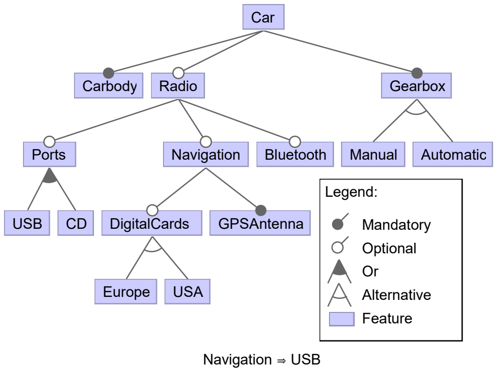

Figure 1: Example feature model adapted from Ananieva et al. [3]

Without the cross-tree constraints, computing the number of valid configurations has linear time complexity in the number of features. Only considering the tree-structure, the selections in a sub-tree are completely independent from selections in other sub-trees. Therefore, it is possible to compute the model count of each sub-tree separately. Without cross-tree constraints the model defines 66 valid configurations. This can be computed by traversing through the tree once recursively by applying rules for each relation type. For example, the number of valid configurations of an alternative sub-tree is equal to the sum of its childrens model counts. Leaf features contain exactly one configuration. Thus, the computation has linear time complexity. However, this is not possible anymore when considering cross-tree constraints.

With cross-tree constraints, the number of configurations cannot be computed in polynomial time complexity. Every feature model can be translated to a propositional formula [30]. Furthermore, a feature model that contains cross-tree constraints can represent every propositional formula [25]. Thus, computing the satisfiability of a model with those constraints is at least as hard as SAT. Determining whether a formula is satisfiable after computing the number of valid assignments is trivial. Therefore, #SAT is at least as hard as SAT [23]. It follows that the complexity of counting the number of valid configurations with cross-tree constraints is not polynomial.

4 MODEL COUNTING

SAT solvers are used to determine whether a propositional formula is satisfiable (i.e., there is an assignment with which the formula evaluates to true) [20]. Let F be a propositional formula and $vars(F)$ with $|vars(F)| = n$ the corresponding set of variables. An assignment is a function $\alpha : vars(F) \rightarrow \{0, 1\}$ that maps variables contained in F to the truth values 0 or 1 [27]. Assignments can be partial, meaning that not every variable $v \in vars(F)$ is mapped to 0 or 1. Otherwise, the assignment is called full [27]. The cardinality $|\alpha| \leq n$ corresponds to the number of variables mapped to 0 or 1 in

α. The function $F(\alpha) \rightarrow \{0, 1\}$ evaluates whether a full assignment α satisfies the formula F. We call an assignment α with $F(\alpha) = 1$ a valid assignment or a model.

Propositional model counting (for short #SAT) is defined as computing the number of valid full assignments of a propositional formula [18, 27]. $\#F = |\{\alpha | F(\alpha) = 1\}|$ corresponds to the number of models (i.e., valid full assignments of formula F).

#SAT is widely believed to be harder to solve than SAT [12]. #SAT is obviously at least as hard as SAT, because it is trivial to determine whether there is at least one solution after computing the number of solutions [23]. Even though the problem is difficult with regard to the theoretical complexity, the #SAT community made significant advances in the last decade [6, 12, 15, 28, 42]. We focus on the major model counting methods, namely DPLL-based [6, 10, 12, 39, 42], d-DNNF-based [15, 28, 32], and BDD-based [43] counting. We give a short introduction to those three methods in the following.

Davis-Putnam-Logemann-Loveland (DPLL)-based counting uses an explorative search through the tree representing the formula with n variables. Each depth in the tree corresponds to the assignment of one variable of the formula. The goal is to find an assignment α with $|\alpha| \leq n$ that either satisfies or does not satisfy the formula for each possible assignment of the remaining $n - |\alpha|$ variables. If the formula evaluates to false under the assignment α, the number of models for the corresponding branch is 0. If it evaluates to true, the number of models for the corresponding branch is $2^{n-|\alpha|}$, which is the number of possible assignments of the remaining variables. A valid full assignment has exactly $2^{n-n} = 1$ model. After computing a result for a branch, DPLL uses backtracking to find remaining assignments. This is done until the number of models for every branch is known. The sum of computed results is the exact number of models [11].

Another possible way to compute number of models are d-DNNF compilers. The term d-DNNF stands for deterministic, decomposable negation normal form. d-DNNF is a subset of NNF that satisfies determinism and decomposability [16]. A formula L is called *deterministic* if each child $D_1, ..., D_n$ of a disjunction $D \in L$ is logically disjunct (i.e., $\forall i, j : i \neq j : D_i \wedge D_j \models \bot$) [16]. A formula is called *decomposable* if the children $C_1, ..., C_n$ of a conjunction C share no variables (i.e., $\forall i, j : i \neq j : vars(C_i) \cap vars(C_j) = \emptyset$) [16]. Determinism implies that the children $D_1, ..., D_n$ of a disjunction D share no common solutions. Therefore, the model count of the sub-tree corresponding to the disjunction is equal to the sum of the children's results (i.e., $\#D = \sum_{i=1}^{n} \#D_i$) [11]. Decomposability implies that each conjunction C can be decomposed into disjoint sub-trees representing the children $C_1, ..., C_n$. It follows that the model count of the conjunction is equal to the product of the results for each subtree (i.e., $\#C = \prod_{i=1}^{n} \#C_i$) [11]. Using both properties, it is possible to compute the overall number of solutions by traversing through the formula once [11]. A d-DNNF-based model counting corresponds to compiling a propositional formula to d-DNNF. A common way to compile a CNF to d-DNNF is an exhaustive DPLL algorithm [15, 32]. After the compilation, computing the model count takes polynomial time [15]. The d-DNNF is strictly more succinct than binary decision diagramms [16]. Therefore, d-DNNFs should take less memory to express a propositional formula.

A binary decision diagram (BDD) is a directed acyclic graph that represents a propositional formula F [19]. Each BDD contains two terminal nodes, with values 0 and 1, and at least one variable node. Every variable node corresponds to one variable $v \in vars(F)$ and has exactly two child nodes, one high and one low child. Each child can either be another variable node or a terminal node. Suppose an assignment α, it can be evaluated by recursively traversing through the BDD. At each variable node corresponding to variable v, the next edge taken by the traversal depends on $\alpha(v)$. Depending on whether $\alpha(v) = 0$ or $\alpha(v) = 1$, the edge to the low or high child is taken respectively. The assignment α satisfies the formula if the resulting path ends in the 1-terminal node [19]. The required space for a BDD is highly dependent on the variable ordering [31, 34]. Therefore, the literature often considers ordered binary decision diagrams (OBDD) and reduced OBDDs [19, 22, 31]. A BDD is called *ordered* if the variable index of a parent is always higher than the indices of its children [17]. Additionally, a BDD is called *reduced* if the two following properties hold. First, there is no variable node whose low and high child are the same node. Second, there are no two distinct nodes for which the ancestors are equivalent [43]. BDDs can be compiled from a propositional formula using an exhaustive DPLL search as d-DNNF [21, 22]. The model count can be determined by traversing through the diagram starting from the 1-terminal node. This improves the runtime as unsatisfied paths are not considered. Furthermore, the efficiency of computing the number of valid assignments can be improved by using dynamic programming [17].

5 EVALUATION

In this section, we report on our evaluation of #SAT solvers on industrial feature models. We examine the scalability of #SAT solvers, the correlation of size of a system with the scalability, and the evolution of configurable systems.

5.1 Evaluated #SAT Solvers

Our benchmark considers three types of model counters. Mainly, we evaluated DPLL-based solvers and CNF to d-DNNF compilers. Additionally, we added one CNF to BDD compiler for comparison, as previous works also used BDDs for model counting [2, 19, 29]. Every chosen solver uses CNFs in DIMACS-format as input and is publicly available. Furthermore, we only evaluated exact #SAT solvers, contrary to approximate ones.

- DPLL-based solvers: Relsat [6], Cachet [39], sharpSAT [42], countAntom [12], picoSAT [10]
- CNF to d-DNNF compilers: c2d [15], d4 [28], dSharp [32]
- CNF to BDD compiler: CNF2OBDD [43]

5.2 Subject Systems

We evaluate the performance of the listed #SAT solvers on industrial configurable systems from the automotive and operating system domain. First, we analyze configurable systems provided by Knüppel et al. [25]. These systems were extracted by snapshots of an automotive product line and translating KConfig and CDL models. KConfig was designed for managing Linux configurations and CDL for managing eCos [25]. The systems are available as FeatureIDE

Subject Systems	#Models	#Features	#Constraints
KConfig	7	96-6467	14-3545
CDL	116	1178-1408	816-956
Automotive02	4	14010-18616	666-1369
Automotive03	5	149-588	0-1184
Automotive04	50	127-531	0-623
Automotive05	136	246-1674	0-11632

Table 1: Number of models, range of features, and range constraints of the evaluated subject systems

feature models.[1] We converted the models to DIMACS-format using FeatureIDE 3.5.5.[2]

Second, we were given access to industrial models for three different systems from the automotive domain. These models were provided in a proprietary format. With the help of company interns, we translated their data structure into feature models. Afterwards, we transformed the resulting models to the DIMACS-format using FeatureIDE 3.5.5. Each feature and constraint in the system included a date which indicated the introduction into the system. Furthermore, some included a date that indicated exclusion. Using these properties, we created snapshots that represented the system only with features and constraints that are included at the given timestamp. In the remainder, we distinguish between the entire system and snapshots of that system. We refer to these snapshots as models. Furthermore, if we use the term system, we always refer to the latest model of that system. Table 1 presents properties of all systems considered in the experiment. All models and systems only contain boolean variables and boolean constraints.

5.3 Experiment Design

In the first stage, we evaluated all solvers specified in Section 5.1 on every subject system described in Section 5.2. If a solver was not able to compute a result within 10 minutes, the process was terminated. In the second stage, we repeated the experiment with all the systems and models that could not be evaluated within 10 minutes by any of the solvers. Here, the timeout was set to 24 hours. For 127 systems, a single solver that does not scale might need more than four months for the computation. Therefore, we excluded all solvers in the second stage that could not evaluate at least half of the models within 10 minutes in the first stage.

All solvers except CNF2OBDD provided parameters to internally limit the maximum memory used. During the entire benchmark, we set this memory limit for each solver to 8 GB. However, sometimes solvers exceeded the given limit. If a solver reached 10 GB of memory, the process was terminated to prevent an unintended use of memory. The solver countAntom allows multi-threading and the maximum number of used threads can be defined. During the first stage, we evaluated countAntom with one and four available threads separately for better comparison. For the second stage, we only considered countAntom with four available threads. During the remainder of the evaluation, we consider the experiment with four threads if not stated otherwise.

5.4 Results

Figure 2 shows the runtime of each solver on all configurable systems listed in Section 5.2. The runtime is logarithmic. The red line indicates the timeout and the blue line indicates that a solver reached the memory limit. In the first stage, Picosat reached the timeout on every system and Relsat for 119 of the 127 (93.7%) systems. CNF2OBDD scaled to only 3 of 127 systems (2.36%), namely axTLS (96 features), uClinux-base (380), and Automotive04 (531). It reached the memory limit of 8 GB for 122 and the timeout for 2 systems. Most other solvers only reached timeout for Linux and Automotive05. The only other exceptions were Cachet and c2d which both had an additional timeout on one other model. countAntom was the fastest solver for 113 systems. countAntom with only one thread was 1.9 times slower on average than with four threads. However, it was still the fastest solver on all these 113 systems. dSharp was terminated for one system for reaching the hard memory limit of 10 GB.

In the second stage, the experiment was repeated without Picosat, Relsat, and CNF2OBDD with a timeout of 24 hours to examine whether a solver is able to compute the remaining systems. However, even within 24 hours not a single solver was able to compute a result for Linux or the latest model of Automotive05.

Figure 3 shows the runtimes in seconds of the six remaining solvers on the 5 Automotive03, 50 Automotive04, and 136 Automotive05 models. In each diagram, the runtime has a logarithmic scale. For Automotive03 and Automotive04, c2d required up to 5.7 seconds to evaluate the models. None of the other solvers remaining in the second stage needed more than 250 milliseconds for any of these models. Cachet and sharpSAT even computed the number of products within 50 milliseconds for the models of Automotive03 and Automotive04. The following solvers were the fastest ones on at least one of these models: sharpSAT (fastest for 29 models), Cachet (24), and dSharp (1).

While the remaining solvers scaled for every model of Automotive03 and Automotive04, this is not the case for Automotive05. The model count of 33 of the 136 models could be computed by at least one solver within 10 minutes. Due a lack of time, the experiment was repeated with a 24 hours timeout with only the three solvers that solved the highest number of models: c2d (33 solved models), countAntom (31/ 23 with one thread), and d4 (29). Within 24 hours, the number of configurations could be computed for 62 models by c2d (51 solved models), countAntom (62), and d4 (49). All of those 62 models were within the 65 earliest ones. For the 65th model 21.6 hours were required to compute the number of configurations.

Figure 4 shows the correlation of the number of features, constraints, and clauses with the runtime of the fastest solver. In each diagram, both scales are logarithmic. Every system with fewer than 1,000 features, 1,000 constraints, or 10,000 clauses was evaluated within 100 milliseconds. The two systems that reached the timeout, the Linux product line and Automotive05, contain 6,467 and 1,663 features. The number of configurations for Automotive02 which contains 18,616 features, 1,369 constraints, and 350,221 clauses was computed within 500 milliseconds.

Figure 5 shows the runtime of the fastest solver for Automotive03, Automotive04, and Automotive05 in correlation to number

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

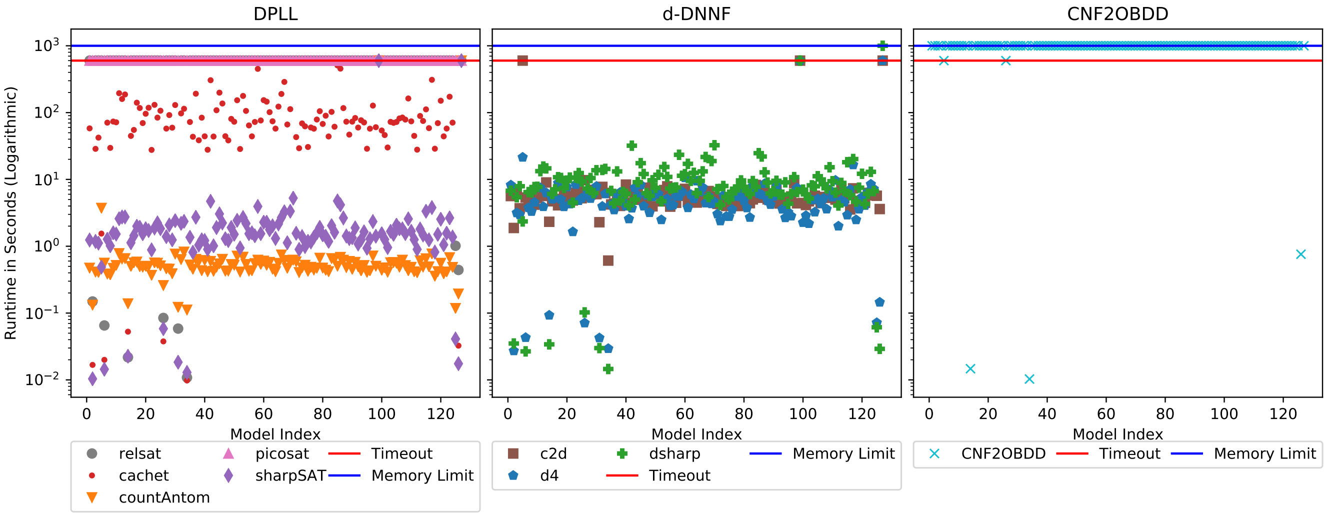

Figure 2: Runtime in seconds (logarithmic) of all solvers on every system with a timeout of 10 minutes

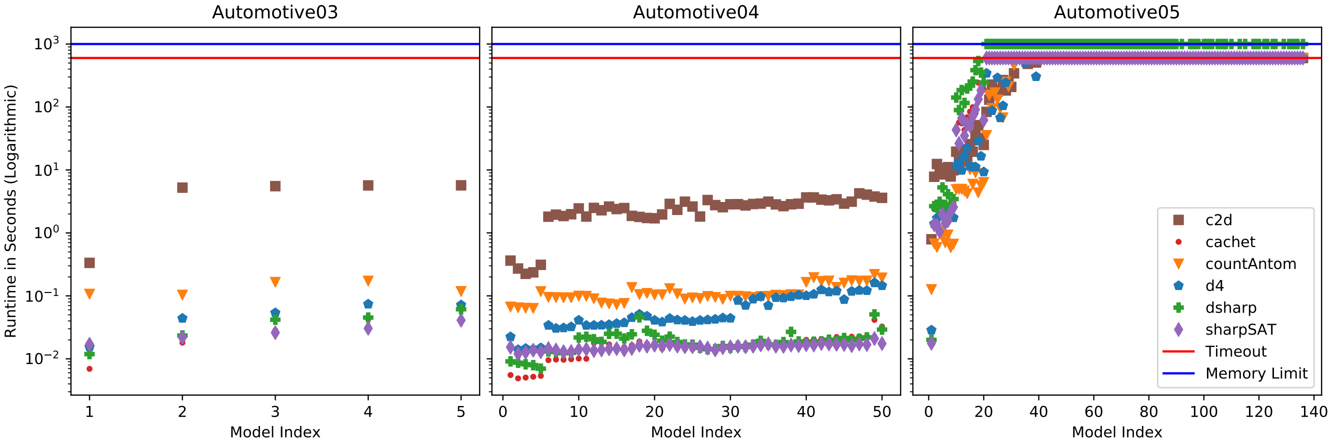

Figure 3: Runtime in seconds of remaining solvers on every model of Automotive03, Automotive04, and Automotive05 with a timeout of 10 minutes

Subject System	#Models(Solved)	#Configurations
KConfig	7(6)	$8.3 * 10^{11} - 2.1 * 10^{201}$
CDL	116(116)	$2.6 * 10^{118} - 3 * 10^{136}$
Automotive02	4(4)	$4.7 * 10^{1260} - 1.7 * 10^{1534}$
Automotive03	5(5)	$5.8 * 10^{28} - 2.5 * 10^{31}$
Automotive04	50(50)	$1.6 * 10^{13} - 4.8 * 10^{23}$
Automotive05	136(62)	$2.2 * 10^{48} - 1.1 * 10^{66}$

Table 2: Number of models and range of number of valid configurations

of features, constraints, and clauses. The three systems are distinguished by means of different colors. In each diagram, the runtime has a logarithmic scale. Every model with less than 1,000 features, 3,000 constraints, or 10,000 clauses was evaluated within 100 milliseconds.

The computed number of products in correlation to the number of features systems are displayed in Figure 6 for every evaluated model. Both scales are logarithmic. The different systems are distinguished by means of different colors. The results of Automotive02 are omitted for better readability. Table 2 shows the ranges of number of configurations of each system, including Automotive02. For every evaluated system that contains multiple models, the number of features is weakly monotonically increasing. This is not the case for the number of configurations as sometimes a later model has fewer configurations. However, overall the configuration space of every automotive system is also growing over time. Table 2 shows the ranges of number of configurations for every evaluated system.

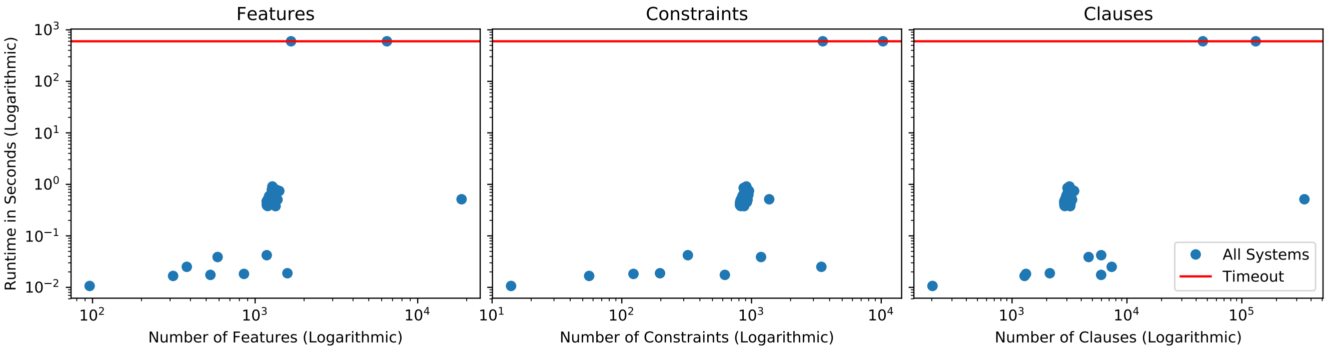

Figure 4: Correlation of number of features, constraints, and clauses with the runtime of the fastest solver on every system

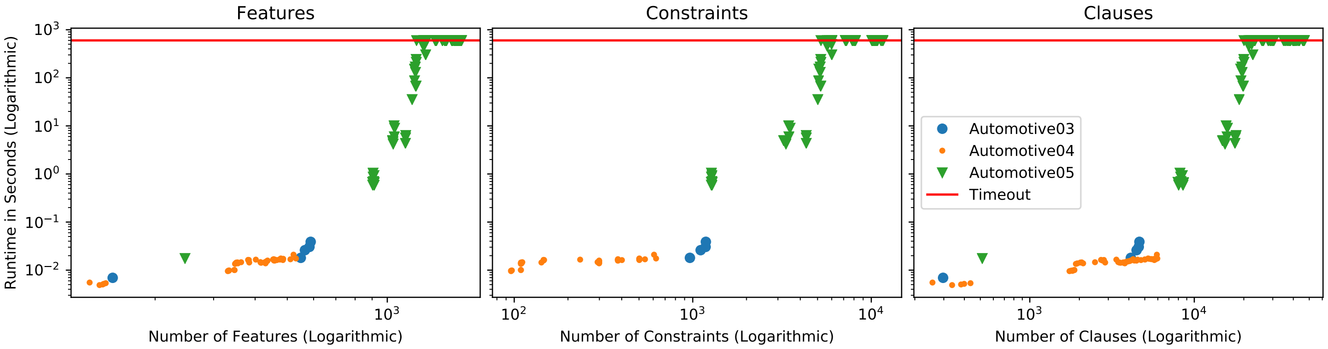

Figure 5: Correlation of number of features, constraints, and clauses with the runtime of the fastest solver for every model of Automotive03, Automotive04, and Automotive05

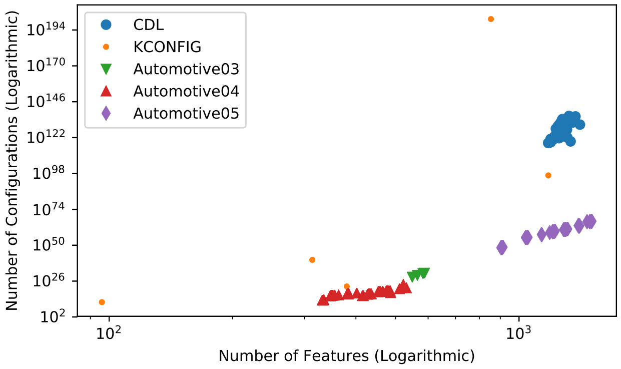

Figure 6: Correlation between number of features and number of configurations for all systems

5.5 Discussion

In this section, we discuss the results regarding our five research questions **RQ1-RQ5**.

RQ1: Do #SAT solvers scale to industrial configuration spaces? Our results indicate that it is possible to apply #SAT solvers to analyze various configurable systems. Overall, the model count was computed for 125 of the 127 analyzed systems and all models of Automotive02-Automotive04 within 10 minutes. However, the current solvers do not scale for all industrial systems that were available to us. Two systems could not be evaluated: Automotive05 and Linux. Out of the 136 Automotive05 models only 35 were evaluated within 10 minutes. With a 24 hours timeout, 62 models were evaluated by at least one solver.

RQ2: Is one #SAT solver superior to the other solvers? None of the solvers was superior to the other solvers in every instance. Cachet is the fastest solver for 24 out of the 55 models from Automotive03 and Automotive04, but only for 7 other systems. c2d solved the highest number of models from Automotive05 of all solvers within 10 minutes, but required by far the longest runtime for the Automotive03 and Automotive04 models of the solvers considered in the second stage. However, countAntom was the fastest one in 113 of the 127 systems that were solved by at least one solver within 10 minutes. Furthermore, the solver computed the highest number of models from Automotive05 within 24 hours.

Even though the best performing solver countAntom is DPLL-based, the results do not indicate a superiority of DPLL-based over d-DNNF-based solvers regarding the scalability of computing the number of configurations. However, for larger systems the BDD-based solver has scalability issues, as the memory limit was reached for 122 systems (i.e., 96.1% of all systems). picoSAT is the only solver that computes the number of valid solutions by enumerating them and scaled to none of the evaluated systems.

RQ3: Does the runtime of the solvers correlate to the size or complexity of the configuration space? The results indicate that it is always easy to compute the number of configurations for small and incomplex systems. For example, every system with less than 1,000 features or constraints was evaluated within 100 milliseconds. However, a large number of features, constraints, or clauses does not necessarily cause a time-consuming computation. The Automotive02 system contains by far the most features, but sharpSAT still evaluated it in less than 500 milliseconds. Outside of the given experiment design, we created a model of Automotive05 that included every feature and constraints that appears in the product line at any point in time. This model contains 27,532 constraints. Still, the fastest solver did not even need 2 seconds for the evaluation, while reaching timeout for models with less than 10,000 constraints.

Nevertheless, the scalability of different versions of a single system seems to positively correlate with the size and complexity. This is indicated by the benchmarks for Automotive03, Automotive04, and Automotive05 in Figure 5.

RQ4: How does the number of valid configurations relate to the number of all configurations? A model with n features contains up-to 2^n configurations [9]. However, a larger n does not guarantee a larger number of valid configurations. Busybox contains 854 features and induces $2.1 * 10^{201}$ valid configurations while the largest model from Automotive05 that could be evaluated contains 1506 features and induces $1.1 * 10^{66}$ valid configurations. Nevertheless, the results for Automotive03, Automotive04, and Automotive05 indicate that within similiar systems there is a positive correlation between all and valid configurations.

RQ5: How does the number of valid configurations change during the evolution of a configurable system? During our experiments only four configurable systems contained multiple models: Automotive02-Automotive05. For each of those systems the number of features monotonically increased at every new model. The results also indicate that configuration spaces of configurable systems continually grow. However, there are cases where a newer model induced fewer configurations. In these cases, the corresponding update introduced new constraints and only few new features.

In order to improve the scalability of #SAT solvers on configurable systems, we consider two possibilities. On the one hand, the #SAT community pushes the limits by further improving solvers. On the other hand, even continually improved solvers might not scale for large configurable systems. Another way to improve scalability, might be to limit the configuration space by reducing the variability (e.g., with feature-model slicing [1, 4, 26] or by modularity principles [40]).

5.6 Threats to Validity

During the evaluation of #SAT solvers on industrial feature models, we identified a few potential threats to validity.

Translating the subject systems to feature models. Our translation of the automotive product lines might be incorrect. However, we created the parser in direct cooperation with company interns. Furthermore, the resulting model was reviewed by them. Another threat is that we could not parse a few constraints as they contained features that did not appear in the feature model. Therefore, we possibly reduced the complexity of the resulting models.

Knüppel et al. also remarked some threats to internal validity regarding their translation of configurable systems [25]. First, there are differences between CDL and KConfig and the target format. Second, the translation may have removed a few cross-tree constraints. Furthermore, a few cases lead to features that did not appear in the input format [25].

Translating feature models to the DIMACS-format. For the translation to DIMACS-format, we used the FeatureIDE-library [24]. In order to ensure a correct translation, we performed the following sanity checks. First, we manually computed the model count of small feature models and compared these results with the ones computed by the solvers. Second, we made changes to the feature model that should change the model count in a certain way. For example, we added an optional feature to the root which should always double the number of products. Another important aspect of the translation to CNF is that the model count of the new formula has to stay the same. This is not given for every conversion method [27]. However, FeatureIDE uses a transformation that does not introduce new variables, nor changes the number of solutions. The results of the sanity checks support this claim.

Computation of the solvers. We only used external solvers without a possibility of directly verifying the results. However, for every computed instance the model count returned by each solver was equal. This is a strong indicator for the correctness of the solvers.

External Validity. We can not claim that our results can be transferred to all other industrial configurable systems. The product lines provided by our industry partner are limited to one domain and company. Furthermore, Knüppel et al. also argued that their results can not directly be transferred to other real-world systems [25]. However, overall we considered multiple domains and various different systems. Therefore, our results possibly indicate the scalability of #SAT solvers for other configurable systems.

6 RELATED WORK

Model Counting of Configurable Systems. Kübler et al. [27] also evaluated the use of two older #SAT solvers, Cachet [38] and c2d [15], on three different body styles of a automotive product line. We evaluated both solvers and they were outperformed by newer solvers on most instances. However, the authors also proposed their own model counter that was not based on conjunctive normal form and performed well even on larger systems. We were not able to acquire their solver or their evaluated product lines. Therefore, we could not directly compare the results.

Pohl et al. [37] evaluated different feature model analyses including model counting using binary decision diagrams, constraint satisfaction problem solvers, and SAT solvers. However, the authors used models with much smaller configuration spaces and fewer features for their evaluation. Their analyzed configuration spaces only reached up to 10^8 valid configurations.

Current Tool Support. BDDs are a popular choice for counting the number of products of configurable systems [2, 19, 29]. The benchmark of Pohl et al. indicated that BDDs might be faster than

#SAT solvers [37]. Furthermore, it is possible to compute the BDD offline and answer queries on the live system in polynomial time [19]. However, our results indicate that BDDs need too much memory for larger feature models. Additionally, d-DNNFs can be computed offline as well and are stricly more succinct than BDDs [16].

FeatureIDE uses a regular SAT solver to compute the number of valid configurations [41]. The tool realizes counting with a regular SAT solver using blocking clauses; after finding a valid assignment α, it is negated and added as a clause to the formula. Thus, α is not a valid assignment for the resulting formula and the next run of the solver returns another assignment. For each found assignment (i.e., valid configuration), an execution from scratch is required. Therefore, the algorithm does not scale for larger systems [43].

Non-propositional Model Counting. Constraint satisfaction problems (CSP) are an alternative to propositional logic for the representation of feature models [7–9, 37]. CSPs are defined by a set of variables, domains for each variable, and constraints over these variables. For CSPs, the variables may also be integers or intervals contrary to propositional boolean variables which are strictly binary [7]. Benavides et al. used constraint programming (CP) to compute the number of valid configurations of feature models. However, the models considered in their experiment only included up to 23 features [9]. The results of the experiment of Pohl et al. which compared SAT solvers, BDDs, and CSP solvers also indicated that the analyzed CSP solvers scale far worse than the #SAT solvers evaluated in our experiment [37]. Munoz examined counting the number of valid configurations of feature models with numerical features for uniform random sampling. The authors evaluated an SMT solver, a CP solver, and the #SAT solver sharpSAT. The numerical values were translated to propositional logic using bit-blasting [33]. In their experiment, sharpSAT vastly outperformed the CP and SMT solver.

7 CONCLUSION

Managing the variability of configurable systems is hard without computing the number of valid configurations. In this paper, we analyzed three d-DNNF-based [15, 28, 32], five DPLL-based [6, 10, 12, 39, 42], and one BDD-based [43] #SAT solver on industrial configurable systems. Those systems consisted of models provided by Knüppel et al. [25] and three automotive product lines provided by our industry partner. Most of the features and constraints of the automotive product lines were only valid for a specific period of time. Therefore, we were able to create snapshots, each representing one timestamp for each product line.

Out of the 127 evaluated systems, 125 could be computed within 10 minutes by at least one solver. However, the remaining two systems, Linux and a majority of the snapshots of the largest automotive product line, could not even be computed within 24 hours. This indicates that current #SAT solvers scale to many, but not to all systems. Furthermore, the results for different snapshots of the same system indicate that the configuration space of these systems is continually growing during the evolution. While the #SAT community keeps pushing the limits for their solvers, even new solvers do not necessarily scale for the growing systems. Deliberately reducing the variability of a configurable system might be necessary to enable the computation of the number of valid configurations.

In our experiments, some solvers were able to evaluate only a minority, if any, of the given systems. While, none of the solvers was strictly superior to all other ones, countAntom [12] was the fastest solver for 89.0% of the evaluated systems. Nevertheless, it might be beneficial to use different solvers in parallel.

To measure the complexity of a system, we considered the number of clauses and constraints. For the size, we considered the number of features. The results indicated that the number of valid configurations of a more complex or larger system does not necessarily take more time to be computed. However, the runtime has a positive correlation with the size and complexity within similar systems. Furthermore, a majority of the solvers scales for all smaller or less complex models.

8 FUTURE WORK

In this section, we describe further tasks in counting the configuration space of configurable systems.

Approximate #SAT Solvers. In this paper, we focused on exact model counting. However, the results indicated that exact model counters may not be scalable for large industrial systems. One solution for this problem might be approximate #SAT solvers [13, 18]. These aim to count the number of satisfying assignments within a given confidence level [13]. Approximate solvers often allow various options to parameterize the process of model counting [13, 18]. For approximated analysis of configurable systems it is necessary to find suitable parameters that result in efficient and effective computations.

Further Metrics for a Meta-Solver. Our results show that the solvers perform differently depending on the system. None of the solvers was strictly superior to the others. Analyzing the meta-data of configurable systems might enable an efficient meta-solver that selects the most promising solver depending on a given instance. For regular SAT, it is already known that selecting a solver based on single instances vastly improves the performance [44].

Directly Translate Feature Models to d-DNNF. Every experiment used propositional formulas in conjunctive normal form. The translation to CNF was not considered in the runtime. However, for the larger systems, the translation requires a considerable amount of time. Directly translating the feature model to d-DNNF might result in two benefits. First, the time overhead of computing the model to CNF would be eliminated. Second, it is possible that the direct translation is even less expensive itself.

Analyze Memory Usage. We mainly focused on the runtime of the solvers. However, memory usage plays an important role as well. Especially, the d-DNNF and BDD compilers need a high amount of memory to translate the formula.

Growth of Configuration Spaces. Our results indicated that the size of configurable systems we analyzed grow over time. However, we do not have enough data to translate this observation on other systems. Analyzing how fast configuration spaces are growing might enable the following estimations. First, the growth might indicate whether #SAT solvers scale for a future model. Second, the number of valid configurations of a future model can possibly be estimated without using a #SAT solver.

REFERENCES

[1] Mathieu Acher, Philippe Collet, Philippe Lahire, and Robert B France. 2011. Slicing feature models. In *Proceedings of the 2011 26th IEEE/ACM International Conference on Automated Software Engineering*. IEEE Computer Society, 424–427.

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

[3] Sofia Ananieva. 2016. *Explaining Defects and Identifying Dependencies in Interrelated Feature Models*. Ph.D. Dissertation. Institute of Software.

[4] Sofia Ananieva, Matthias Kowal, Thomas Thüm, and Ina Schaefer. 2016. Implicit constraints in partial feature models. In *Proceedings of the 7th International Workshop on Feature-Oriented Software Development*. ACM, 18–27.

[5] Ebrahim Bagheri, Tommaso Di Noia, Dragan Gasevic, and Azzurra Ragone. 2012. Formalizing interactive staged feature model configuration. *Journal of Software: Evolution and Process* 24, 4 (2012), 375–400.

[6] Roberto J Bayardo Jr and Joseph Daniel Pehoushek. 2000. Counting models using connected components. In *AAAI/IAAI*. 157–162.

[7] David Benavides, Sergio Segura, and Antonio Ruiz-Cortés. 2010. Automated analysis of feature models 20 years later: A literature review. *Information Systems* 35, 6 (2010), 615–636.

[8] David Benavides, Sergio Segura, Pablo Trinidad, and Antonio Ruiz-Cortés. 2005. Using Java CSP solvers in the automated analyses of feature models. In *International Summer School on Generative and Transformational Techniques in Software Engineering*. Springer, 399–408.

[9] David Benavides, Pablo Trinidad, and Antonio Ruiz-Cortés. 2005. Automated reasoning on feature models. In *International Conference on Advanced Information Systems Engineering*. Springer, 491–503.

[10] Armin Biere. 2008. PicoSAT essentials. *Journal on Satisfiability, Boolean Modeling and Computation* 4 (2008), 75–97.

[11] Armin Biere, Marijn Heule, and Hans van Maaren. 2009. *Handbook of satisfiability*. Vol. 185. IOS press.

[12] Jan Burchard, Tobias Schubert, and Bernd Becker. 2015. Laissez-faire caching for parallel# SAT solving. In *International Conference on Theory and Applications of Satisfiability Testing*. Springer, 46–61.

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

[14] Paul C Clements, John D McGregor, and Sholom G Cohen. 2005. *The structured intuitive model for product line economics (SIMPLE)*. Technical Report. CARNEGIE-MELLON UNIV PITTSBURGH PA SOFTWARE ENGINEERING INST.

[15] Adnan Darwiche. 2004. New advances in compiling CNF to decomposable negation normal form. In *Proceedings of the 16th European Conference on Artificial Intelligence*. Citeseer, 318–322.

[16] Adnan Darwiche and Pierre Marquis. 2002. A knowledge compilation map. *Journal of Artificial Intelligence Research* 17 (2002), 229–264.

[17] David Fernandez-Amoros, Ruben Heradio, Jose A Cerrada, and Carlos Cerrada. 2014. A scalable approach to exact model and commonality counting for extended feature models. *IEEE Transactions on Software Engineering* 40, 9 (2014), 895–910.

[18] Carla P Gomes, Ashish Sabharwal, and Bart Selman. 2006. Model counting: A new strategy for obtaining good bounds. In *AAAI*. 54–61.

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

[20] Ruben Heradio, David Fernandez-Amoros, Jose A Cerrada, and Ismael Abad. 2013. A literature review on feature diagram product counting and its usage in software product line economic models. *International Journal of Software Engineering and Knowledge Engineering* 23, 08 (2013), 1177–1204.

[21] Jinbo Huang and Adnan Darwiche. 2004. Using DPLL for efficient OBDD construction. In *International Conference on Theory and Applications of Satisfiability Testing*. Springer, 157–172.

[22] Jinbo Huang and Adnan Darwiche. 2005. DPLL with a trace: From SAT to knowledge compilation. In *IJCAI*, Vol. 5. 156–162.

[23] David S Johnson. 1992. The NP-completeness column: an ongoing guide. *Journal of algorithms* 13, 3 (1992), 502–524.

[24] Christian Kastner, Thomas Thum, Gunter Saake, Janet Feigenspan, Thomas Leich, Fabian Wielgorz, and Sven Apel. 2009. FeatureIDE: A tool framework for feature-oriented software development. In *Proceedings of the 31st International Conference on Software Engineering*. IEEE Computer Society, 611–614.

[25] Alexander Knüppel, Thomas Thüm, Stephan Mennicke, Jens Meinicke, and Ina Schaefer. 2017. Is there a mismatch between real-world feature models and product-line research?. In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*. ACM, 291–302.

[26] Sebastian Krieter, Reimar Schröter, Thomas Thüm, Wolfram Fenske, and Gunter Saake. 2016. Comparing algorithms for efficient feature-model slicing. In *Proceedings of the 20th International Systems and Software Product Line Conference*. ACM, 60–64.

[27] Andreas Kübler, Christoph Zengler, and Wolfgang Küchlin. 2010. Model counting in product configuration. *arXiv preprint arXiv:1007.1024* (2010).

[28] Jean-Marie Lagniez and Pierre Marquis. 2017. An Improved Decision-DNNF Compiler.. In *IJCAI*. 667–673.

[29] Marcilio Mendonca, Moises Branco, and Donald Cowan. 2009. SPLOT: software product lines online tools. In *Proceedings of the 24th ACM SIGPLAN conference companion on Object oriented programming systems languages and applications*. ACM, 761–762.

[30] Marcilio Mendonca, Andrzej Wąsowski, and Krzysztof Czarnecki. 2009. SAT-based analysis of feature models is easy. In *Proceedings of the 13th International Software Product Line Conference*. Carnegie Mellon University, 231–240.

[31] Marcilio Mendonca, Andrzej Wasowski, Krzysztof Czarnecki, and Donald Cowan. 2008. Efficient compilation techniques for large scale feature models. In *Proceedings of the 7th international conference on Generative programming and component engineering*. ACM, 13–22.

[32] Christian Muise, Sheila McIlraith, J Christopher Beck, and Eric Hsu. 2010. Fast d-DNNF Compilation with sharpSAT. In *Workshops at the twenty-fourth AAAI conference on artificial intelligence*.

[33] Daniel-Jesus Munoz, Jeho Oh, Mónica Pinto, Lidia Fuentes, and Don Batory. 2019. Uniform random sampling product configurations of feature models that have numerical features. In *Proceedings of the 23rd International Systems and Software Product Line Conference-Volume A*. ACM, 39.

[34] Macha Nikolskaia and Antoine Rauzy. 1998. Heuristics for BDD handling of sum-of-products formulae. In *Proceedings of the European Safety and Reliability Association Conference, ESREL'98*. Citeseer.

[35] Jeho Oh, Don Batory, Margaret Myers, and Norbert Siegmund. 2017. Finding near-optimal configurations in product lines by random sampling. In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*. ACM, 61–71.

[36] Jeho Oh, Paul Gazzillo, Don Batory, Marijn Heule, and Maggie Myers. 2019. Uniform sampling from kconfig feature models. *The University of Texas at Austin, Department of Computer Science, Tech. Rep. TR-19-02* (2019).

[37] Richard Pohl, Kim Lauenroth, and Klaus Pohl. 2011. A performance comparison of contemporary algorithmic approaches for automated analysis operations on feature models. In *Proceedings of the 2011 26th IEEE/ACM International Conference on Automated Software Engineering*. IEEE Computer Society, 313–322.

[38] Tian Sang, Fahiem Bacchus, Paul Beame, Henry A Kautz, and Toniann Pitassi. 2004. Combining Component Caching and Clause Learning for Effective Model Counting. *SAT* 4 (2004), 7th.

[39] Tian Sang, Paul Beame, and Henry Kautz. 2005. Heuristics for fast exact model counting. In *International Conference on Theory and Applications of Satisfiability Testing*. Springer, 226–240.

[40] Reimar Schröter, Sebastian Krieter, Thomas Thüm, Fabian Benduhn, and Gunter Saake. 2016. Feature-model interfaces: the highway to compositional analyses of highly-configurable systems. In *Proceedings of the 38th International Conference on Software Engineering*. ACM, 667–678.

[41] Thomas Thum, Christian Kastner, Sebastian Erdweg, and Norbert Siegmund. 2011. Abstract features in feature modeling. In *2011 15th International Software Product Line Conference*. IEEE, 191–200.

[42] Marc Thurley. 2006. sharpSAT–counting models with advanced component caching and implicit BCP. In *International Conference on Theory and Applications of Satisfiability Testing*. Springer, 424–429.

[43] Takahisa Toda and Takehide Soh. 2016. Implementing efficient all solutions SAT solvers. *Journal of Experimental Algorithmics (JEA)* 21 (2016), 1–12.

[44] Lin Xu, Frank Hutter, Holger H Hoos, and Kevin Leyton-Brown. 2008. SATzilla: portfolio-based algorithm selection for SAT. *Journal of artificial intelligence research* 32 (2008), 565–606.