Comparing Solver Representations for Analyzing Cardinality-Based Feature Models

Fabian Eger
Karlsruhe Institute of Technology
Karlsruhe, Germany
fabian.eger@kit.edu

Lukas Güthing
Karlsruhe Institute of Technology
Karlsruhe, Germany
lukas.guething@kit.edu

Kevin Feichtinger
Karlsruhe Institute of Technology
Karlsruhe, Germany
kevin.feichtinger@kit.edu

Ina Schaefer
Karlsruhe Institute of Technology
Karlsruhe, Germany
ina.schaefer@kit.edu

Abstract

The variability of product lines can exceed purely Boolean configuration spaces. Cardinality-based Feature Models (CFMs) are employed to model multi-instantiation of features along with individually configurable subtrees. Due to the added complexity, the analysis of CFMs cannot be done with state-of-the-art, SAT-based tooling for analyzing Boolean Feature Models (FMs). Analyses on FMs include checking for satisfying configurations, dead features, false optional features, and whether specific configurations are valid according to the FM. In this work, we compare different solver encodings to enable analysis for CFMs. First, we generalize the analyses on Boolean FMs to the notion of cardinalities and the new anomalies that can occur. Second, we present three different mathematical encodings of CFMs for automated reasoning using solvers. Third, we implement the encoding for ILP, SMT, and CSP solvers. We evaluate the feasibility and performance of our encodings on current ILP, SMT, and CSP solvers. Our evaluation shows that our encoding for CSP solvers enables all common analyses with the best performance among the compared encodings and solvers.

CCS Concepts: • **Software and its engineering → Software product lines**.

Keywords: Software product lines, cardinality-based feature models, analysis, comparison.

ACM Reference Format:
Fabian Eger, Lukas Güthing, Kevin Feichtinger, and Ina Schaefer. 2026. Comparing Solver Representations for Analyzing Cardinality-Based Feature Models. In *Proceedings of the 25th ACM SIGPLAN International Conference on Generative Programming: Concepts and Experiences (GPCE '26), June 29, 2026, Brussels, Belgium.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3814885.3816410

GPCE '26, Brussels, Belgium

ACM ISBN 979-8-4007-2718-4/2026/06
https://doi.org/10.1145/3814885.3816410

1 Introduction

Product Line Engineering (PLE) fosters systematic reuse of assets to mitigate issues that come with non-systematically developed product families, using, for instance, clone-and-own [1, 23]. A central aspect of PLE is the explicit modeling of commonalities and differences between products using variability models, like Feature Models (FMs) [4, 9, 21]. FMs describe a system's variability in a tree-like structure that makes the hierarchy between configuration options (*features*) explicit. Addtionally FMs allow to describe the relations between features [1, 18, 23] using constraints. Therefore, a FM reflects the configuration options, which are selectable or deselectable, and their interdependencies. FMs not only make a system's variability explicit and understandable, but they can also be used to analyze dependencies between features and find potential errors in the FM, called *anomalies* [3, 14, 21]. FM anomalies can be, for example, a non-satisfiable FM, i.e., a FM overconstrained to the point that there is no possible valid configuration, or single features that cannot be selected in any valid configuration (*dead features*) [3, 11, 24]. Analyses on FMs are typically done by translating the FM to a Boolean formula and using Boolean Satisfiability Problem (SAT)-solvers for automated reasoning [1, 30]. FM analyses are well-researched and SAT-based solutions have been optimized for product-line analysis [3, 14, 21, 30].

For advanced applications, like distributed systems, cloud computing, or Cyber-Physical Systems (CPSs), the notion of features in FMs lacks expressiveness. Often, features (think of storage or computation cores) can not only be selected or deselected, but also the number of their instances can be configured. For that, Cardinality-based Feature Models (CFMs) include cardinalities, i.e., the possibility to instantiate features multiple times in a variant [10]. Each feature instance also has an individually configurable subtree, so that instances can differ in their respective configurations. While there are optimized approaches to analyze Boolean FMs [31], CFMs currently lack proper automated reasoning and analysis methods [17]. The added complexity introduced by multi-instantiation makes SAT-based approaches insufficient, and other reasoners must be employed [33, 35].

Thus, in this work, we present mathematical formalizations of CFMs that we translated for three different types of solvers. This formalization enables not only analyses of

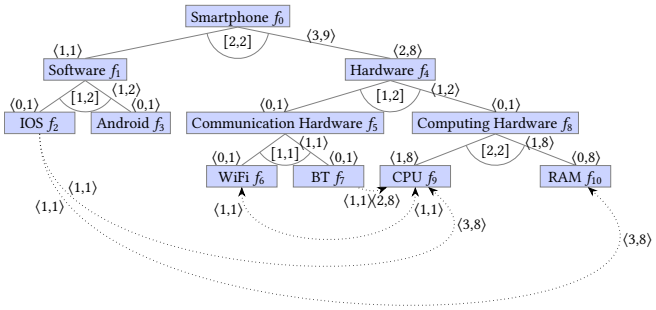

Figure 1. CFM of a Smartphone product line.

CFMs but also performance comparisons between the solvers. As not every representation can capture the CFM's full configuration space without information loss, we investigate the feasibility of different encodings for CFM analyses. We propose translation rules for CFM concepts into different solver-oriented representations and implement those rules, translating CFMs into Integer Linear Program (ILP), Satisfiability Modulo Theories (SMT), and Constraint Satisfaction Problem (CSP) solver representations, and introduce and apply analysis operations for CFM on the presented solver encodings. Because different solvers are better suited for different tasks, we compare their computational effort using one state-of-the-art solver for each solver type. In our evaluation, we compare the performance in terms of analysis time of different solver encodings and discuss their differences. Ultimately, we aim at figuring out which encodings perform best for given analyses and what CFM characteristics influence their performance.

2 Foundations

This section introduces CFMs, their syntax and semantics, and gives an overview of solver types we implemented our encodings for, to automatically reason on CFMs.

2.1 Cardinality-Based FMs

CFMs are a generalization of FODA FMs [6] used to describe Boolean variability. Figure 1 shows an example CFM. It describes a smartphone product line with different operating systems and configurable hardware specs. CFMs have a tree-like structure, with hierarchical parent-child relationships between features. Each feature's multiplicity, i.e., the number of times it can be instantiated per parent feature, is indicated by its *feature instance cardinality*. In Figure 1, this is represented as an interval of the form $\langle l, u\rangle$ directly above the respective feature, where l is the lower and u is the upper bound. For example, in every *Smartphone*, there can be two to eight *Hardware* components. Similarly, one instance *Computing Hardware* can have none to eight instances of *RAM*.

Sibling features are constrained in their instantiations by generalization of the classic *alternative* and *or* relations. Namely, the two cardinalities describing the valid numeric range instantiations of sibling features are called *Group Instance Cardinality* and *Group Type Cardinality*. *Group Instance Cardinality* constrains the number of child instances of all siblings under a parent. It is depicted as an interval $\langle l, u\rangle$ again, but on the rightmost edge of a group. For example, each instance of *Smartphone* must have at least 3 and at most 9 instances of either *Software* or *Hardware*, while *Hardware* itself can only have one or two child instances. *Group Type Cardinality* determines which type the child instances of a parent can be. Specifically, they define a lower and an upper limit to the number of different types of siblings that can be instantiated and are given as $[l, u]$ in Figure 1. For example, every *Smartphone* has to have children of exactly two types, which means that each Phone has to have *Software* as well as *Hardware*. The feature *Communication Hardware*, however, can only have children of exactly one type, i.e., either *WiFi* or *BT*, but not both. This is equal to the *Alternative* group known from Boolean FMs.

Cross-Tree Constraints (CTCs) are less expressive in CFMs compared to Boolean FMs. In CFMs, only *require* or *exclude* constraints are available. A *require* constraint is a generalization of an implication between two features in Boolean FMs. In Figure 1, *require* constraints are depicted as single-headed arrows with a source cardinality interval and a target cardinality interval. For example, the arrow between *IOS* and *RAM* means that if there is exactly one instance of *IOS*, there must be at least three and at most eight instances of *RAM*. An *exclude* constraint forbids two different features to be in defined multiplicities at the same time in a valid configuration. Graphically, this relation is expressed as a double-headed arrow. In Figure 1, if there is exactly one instance of *WiFi*, there can't be exactly one instance of *CPU*, and vice versa.

Compound Intervals are intervals with gaps. As an example, a compound interval could look like $\langle 0, 2\rangle\langle 5, 10\rangle$, which describes any multiplicity between zero and two *or* between five and ten as valid. Compound intervals can either occur directly through constraints while modeling a CFM, or after analysis as a symptom of an existing anomaly. Cardinalities and cardinality intervals in CFMs are interpreted differently in the literature [22]. In our case, *Feature instance*, *group instance*, and *group type cardinalities* are all interpreted *locally*, meaning that all cardinalities have to be satisfied per parent instance. In contrast, the *global* interpretation would interpret every cardinality for the whole CFM, i.e., every *Smartphone* could only have a total of one to eight *CPUs* according to the CFM in Figure 1. The only cardinalities we interpret globally are those appearing in CTCs, as multiplicities relative to parent features cannot be consistently specified across the tree for features with different parents. To estimate upper limits of multiplicities in a CFM, a sufficiently large so-called *Big-M* value is employed [35]. Big-M is calculated by multiplying the upper limits of all cardinalities of a subtree in a CFM. For unbound CFMs, it serves as an approximation to replace 'infinities' and make the respective

model analyzable, as unboundedness often makes reasoning infeasible or impossible. For this work, we will use the Big-M value as a metric for the complexity of a CFM, measuring its potential number of feature instances.

2.2 Solver Categories

For the analysis, we use three solver categories and create encodings to accommodate each solver's requirements.

First, *ILP* solvers, which are mathematical optimization problems where variables are restricted to integers and the constraints are restricted to linear expressions. Additionally, it comprises a linear objective function that specifies the parameters to be optimized, and a solution that is either a maximum or a minimum, depending on the problem being modeled. The constraints of the problem are modeled with linear equalities or linear inequalities involving integer variables. Non-linear constraints can not be directly represented in an ILP solver. We chose the Python library *ortools.linear_solver* from Google's Or-Tools [16] as our ILP solver. Or-Tools' linear optimization library offers different solver backends. For instance, their Mixed Integer Program (MIP) solver, which can be used without a license. Further, the library also enables plugging in licensed solvers for certain problem types. However, this might introduce solver-specific constraints, thereby limiting generalizability. Thus, we use the MIP solver for our evaluation.

Second, *SMT* solvers, which are an extension of SAT solvers. SMT solvers have the richest variable type set of the three solver categories, allowing formulas to contain boolean variables, real and integer numbers, and expressions over strings [5]. An SMT solver is a basic SAT solver, extended with a theory solver that supports linear arithmetic and algebraic data types. The constraints in an SMT solver are restricted to quantifier-free first-order logic over theory atoms. Consequently, SMT solvers have the most expressive constraint and variable set of the three used solver categories. The specific SMT solver that we use in the evaluation is the Z3 solver [12] with the SMT-Lib standard [2]. We utilize the SMT-Lib because it is a widely accepted, more direct, yet exchangeable interface for SMT solvers. In theory, Z3 offers specific optimizations, called tactics, that can be used. However, these tactics heavily depend on the specifics of an encoding, which usually are not in line with the SMT-Lib standard. For our work, we aim to provide a generic encoding that other SMT solvers beyond Z3 can utilize. Thus, we chose Z3, one of the most well-known SMT solvers, for our evaluation, but implemented an encoding based on SMT-Lib for solver exchangeability.

Third, *CSP* is an approach "to find values for a (finite) set of variables satisfying a (finite) set of constraints" [32]. Contrary to propositional logic and ILP, CSP provides a much richer set of high-level modeling elements, i.e., *all-different*, that enable the definition of more complex combinatorial problems with high-level abstractions. Additionally, CSP allows for both linear and non-linear constraints over an arbitrary number of variables, which can take finite and infinite discrete numerical values, not only boolean values [3]. Consequently, CSP solvers support the most abstract constraints of the three categories and are the most flexible considering the finite and infinite domains. Similar to the first solver category, our choice was the Python library *ortools.sat.python.cp_model* from Google's Or-Tools [15]. The Or-Tools CP-SAT solver performs well in community challenges such as the *MiniZinc* challenge [29] and has outperformed other solvers like the *Choco-Solver* for several years.

3 Solver-Oriented Representations of CFMs

Here we explain the different solver-oriented representations of CFMs used in this paper. We need these representations to use the various solvers to analyse the CFMs for anomalies and where those arise. Note that we present the solver representations only in an abstract way. Our reason is that we aim to use abstract representation by mathematical formula in order to explain the different approaches of CFM representations, which are later adapted to the specific solver commands. Our representation of the CFM constraints through equalities and inequalities enables us in the implementation to adapt the constraints to specific solver encodings for CSP, ILP, and SMT, which we then implemented. Our full presentation of all encodings as well as their implementation is available in our replication package [13].

The three different approaches to represent the CFMs used in this paper are Basic Cloning [20], Cloning improved with Integer Leaves, and Multi-Set[26]. The solver representations of each approach are explained below, one subsection for each approach. For the following listings of the abstractions, we denote each parameter with subscript *f*, if the feature is represented (e.g. $Smartphone_f$), and with subscript *cl*, if the clone of a feature is represented (e.g. $Smartphone_{cl}$). Furthermore, we use *p* as an abbreviation for parent.

3.1 Representation using Basic Cloning

The basic cloning approach (also called instance-based approach [17]) creates a constant for every clone (or respective instance) of a feature [20]. In our running example (Figure 1), one constant is created to represent the clone of feature *Software*, and eight constants are created to represent each clone for feature *Hardware*. To ensure correct representation, we add the necessary constraint to each clone. Although in doing so, large CFMs with large cardinality intervals will generate many constants and constraints, the approach establishes a one-to-one relationship between a clone and a Boolean constant because now every constant represents the activity status of a clone. By these means, the solver is able to find valid configuration instances of every declared CFM with local interpretation of the cardinalities [20].

$$\lambda_l^{FI}(k) * p_{cl} \leq \sum_{c_{cl} \in clones(k_f, p_{cl})} c_{cl} \leq \lambda_u^{FI}(k) * p_{cl} \quad (1)$$

To represent *Feature Instance Cardinality* of feature k in a CFM, we use the inequalities outlined in Equation 1. The feature instance interval of feature k is represented by $\lambda_{l/u}^{FI}(k)$, where l represents the lower bound and u represents the upper bound. In this way, the sum of all clones of feature k under the parent clone ($\sum_{c_{cl} \in clones(k_f, p_{cl})} c_{cl}$) is restricted to the upper and lower bounds of the feature instance cardinality multiplied by the constant of the parent clone. The reason for the multiplication is that, if the parent clone is not active, then the sum of all clones of feature k under the parent clone must be zero. Conversely, if the parent clone is active, the left and right sides simplify to the upper and lower bounds, since the lower and upper bounds are multiplied by 1, which represents the parent clone's activation status. Consequently, the validity of the feature instance cardinality of feature k under the parent clone is checked. In our running example (Figure 1), the feature instance cardinalities of feature *Hardware* are represented equationally as $f_0(1) * 2 \leq \sum_{c_{cl} \in clones(f_4, f_0(1))} c_{cl} \leq f_0(1) * 8$. The parent clone is represented by $f_0(1)$, which represents the first clone of the feature *Smartphone* and $\sum_{c_{cl} \in clones(f_4, f_0(1))} c_{cl}$representing the sum of all clones of feature *Hardware* with the first clone of feature *Smartphone* as their parent.

$$\lambda_l^{GI}(k) * p_{cl} \leq \sum_{c_{cl} \in children(p_{cl})} c_{cl} \leq \lambda_u^{GI}(k) * p_{cl} \quad (2)$$

Group Instance Cardinalities are represented similarly, as will be apparent from Equation 2. In the center of the two inequalities, the sum of all direct children clones under the parent clone ($\sum_{c_{cl} \in children(p_{cl})} c_{cl}$) is calculated and then checked against the lower and upper bounds of the group instance cardinality ($\lambda_{l/u}^{GI}(k)$) multiplied by the parent clone constant. The multiplication is used for the same purpose as in the feature instance cardinality. In our running example (Figure 1), the group instance constraint of the root feature is represented equationally as $3 * f_0(1) \leq \sum_{c_{cl} \in children(f_0(1))} c_{cl} \leq 9 * f_0(1)$, where 9 represent the upper bound ($\lambda_{l/u}^{GI}$) and where 3 represents the lower bound of the parent clone $f_0(1)$ on which the group instance cardinality was declared.

$$\lambda_l^{GT}(k) * p_{cl} \leq \sum_{c_f \in children(p_f)} t_{c_f, p_{cl}} \leq \lambda_u^{GT}(k) * p_{cl} \quad (3)$$

For *Group Type Cardinalities*, we iterate over all child features of the parent clone ($\sum_{c \in children(p)} t_{c,p}$) in order to obtain the activity status of each child feature. To begin, for each child feature in Equation 3, we evaluate the predicate $t_{child_f, p_{cl}}$, where *child* represents the child feature and *parent* represents the parent clone. Next, if the parent clone is active, we compare the sum of the evaluated predicates to the lower and upper bounds of the group type constraints. The predicate $t_{child_f, p_{cl}}$ (shown below in Equation 4) returns 1 if at least one clone under the parent clone is active; otherwise, the predicate returns 0. Hence, the sum over all predicates represents the number of child features where there is at least one active clone under the parent clone.

$$t_{child_f, p_{cl}} = \begin{cases} 1 & , (\sum_{c_{cl} \in clones(child_f, p_{cl})} c_{cl}) \geq 1 \\ 0 & , (\sum_{c_{cl} \in clones(child_f, p_{cl})} c_{cl}) < 1 \end{cases} \quad (4)$$

In our running example (Figure 1), the group type cardinality of the root feature *Smartphone* (`[2,2]`) restricts that at least one clone of each child feature must be active. This restriction is represented equationally as follows: $2 * f_0(1) \leq \sum_{c_f \in children(f_0(1))} t_{c_f, f_0(1)} \leq 2 * f_0(1)$. In this case, the sum is represented as $t_{f_1, f_0(1)} + t_{f_4, f_0(1)}$ because these are the only child features (*Software* and *Hardware*) of the root feature *Smartphone*.

$$\left(\sum_{c_{cl} \in clones(l_f)} c_{cl} \right) \in \lambda^{RC}(l) \implies \left(\sum_{c_{cl} \in clones(r_f)} c_{cl} \right) \in \lambda^{RC}(r) \quad (5)$$

Cross-Tree Constraints, such as requires and excludes, are defined on the global count of clones of a feature; therefore, we use the predicate $clones(k_f)$ to represent all clones of feature k, in the whole CFM. Consequently, for the requires constraint in Equation 5, we need to add the following implication to the solver: If the sum of all clones of the left feature in the requires constraints (l_f) is within the left requires interval ($\lambda^{RC}(l)$), then the sum of all clones of the right feature (r_f) must be in the right interval ($\lambda^{RC}(r)$). In our running example (cf. Figure 1), the requires constraint from feature *IOS* to feature *CPU* is represented equationally as: $(\sum_{c_{cl} \in clones(f_2)} c_{cl}) \in [1,1] \implies (\sum_{c_{cl} \in clones(f_{10})} c_{cl}) \in [3,8]$.

$$\neg \left(\left(\sum_{c_{cl} \in clones(l_f)} c_{cl} \right) \in \lambda^{EC}(l) \wedge \left(\sum_{c_{cl} \in clones(r_f)} c_{cl} \right) \in \lambda^{EC}(r) \right) \quad (6)$$

The excludes constraints checks in a manner analogous to the requires constraint. For both features in the constraints, the excludes constraint checks whether the sum total of all active clones of the feature falls within the interval ($\sum_{c_{cl} \in clones(l_f / r_f)} c_{cl} \in \lambda^{RC}(l/r)$). Thus, to encode the behavior of the excludes constraint (as shown above in Equation 6), we add two checks in a conjunction so that only one of the sums may be in its given interval. After this, we negate the conjunction. The only excludes constraint in our running example (Figure 1) is between the features *WiFi* and *CPU*, and this is represented equationally as $\neg((\sum_{c_{cl} \in clones(f_6)} c_{cl}) \in [1,1] \wedge (\sum_{c_{cl} \in clones(f_7)} c_{cl}) \in [1,1])$.

3.2 Modified Cloning Representation using Integer Leaves

Our modification of the basic cloning approach builds on this essential fact: In common solvers, Boolean constants can be created, but so can integer constants too. Of course, we recognize that if we create integer constants for each clone feature in the tree hierarchy, consequently, we will lose the advantage of creating a valid configuration. For example, if we exchange the set of cloned Boolean constants with one integer constant of a feature – where that constant has (a) a feature instance interval maximum greater than one, and (b) one or more children – then we also lose all information about under which clone of the feature it is that the children are or are not active. This, in fact, is the outcome of the multi-set approach (detailed below in Section 3.3).

For these reasons, we replace the cloned Boolean constants with one integer constant *only in cases where a feature is a leaf feature*. Just consider, the multiplicity of features above a leaf feature means that leaf features almost always carry the largest number of Boolean constants. Therefore, by creating an integer constant for each leaf feature (i.e., the number of active leaf clones), we are able to save an immense number of constants. In our running example (Figure 1), the basic cloning approach creates 22 Boolean constants, whereas the integer leaf approach creates only 18 integer constants. Because the leaf features *CPU* and *RAM* add two Boolean constants to every parent clone of the feature *Computing Hardware*, the same leaf features add only one integer constant.

$$t_{child_f,p_{cl}} = \begin{cases} 1 & , child_f \geq 1 \wedge child_f == int \\ 0 & , child_f < 1 \wedge child_f == int \\ 1 & , (\sum_{c_{cl} \in clones(child_f,p_{cl})} c_{cl}) \geq 1 \\ 0 & , (\sum_{c_{cl} \in clones(child_f,p_{cl})} c_{cl}) < 1 \end{cases} \tag{7}$$

This change does not affect the structure of the constraints requires and excludes, feature instance, or group instance cardinality, because actually, this is a solver-side problem of encoding the sum of Boolean and integer constants. However, to encode the group type cardinality of cloning with integer leaves, we need to change the helper predicate (as shown above in Equation 7). Here, two cases must be distinguished. First, if the feature is a leaf and therefore encoded into a single integer constant, then the predicate returns 1 if the constant is greater than or equal to one; otherwise, the predicate returns 0. Second, if the feature is not a leaf and therefore may have more than one Boolean constant (i.e., one for each possible clone), then we have the following: To begin, the group type helper predicate sums up all the clones of the feature under the parent clone ($\sum_{c_{cl} \in clones(child_f,p_{cl})} c_{cl}$), and then the helper checks that sum against the two cases mentioned earlier, greater than or equal to 1, or 0.

3.3 Representation using Multi-sets

Using cloning with integer leaves, we still obtain a substantial number of constants and assertions. For that reason, we use multi-set encoding to try to reduce the required constraints and constants.

Weckesser et al. [35] proposed the multi-set approach, which reduces the necessary number of constants and assertions [26, 35]. The authors defined the multi-set representation by collapsing all feature clones of a feature into a single constant, representing the total number of instances of this feature in a valid configuration. They employ a global interpretation of cardinalities. This is our reason for modifying some of their proposed constraints for the multi-set solver representation with local interpretation.

$$\lambda_l^{FI}(k) * p_f \leq k_f \leq \lambda_u^{FI}(k) * p_f \tag{8}$$

As shown here in Equation 8, to locally interpret the feature instance cardinality, we compare the constant k_f, representing the number of clones of the feature, to the feature instance interval bounds $\lambda_{l/u}^{FI}(k)$ multiplied by the number of parent instances. The multiplication is necessary for the local interpretation, because under each parent clone, it is possible to have the minimum and maximum numbers of features within the feature instance interval. For example, when considering feature *IOS*, the feature instance interval lower bound is 0 and the upper bound is 1. The effect is this constraint: $0 * f_1 \leq f_2 \leq 1 * f_1$. Hence, we allow compound intervals, but otherwise the feature instance cardinality constraint in Equation 8 is equal to the feature instance cardinality constraints in Weckesser et al. [35]. In our case, then, for every interval in the compound interval, the feature instance cardinality equation is added, and f_k needs to fulfill only one of the equations.

$$\lambda_l^{GI}(p) * p_f \leq \sum_{c_f \in children(p_f)} c_f \leq \lambda_u^{GI}(p) * p_f \tag{9}$$

The difference between our group instance cardinality constraint (as shown above in Equation 9) and that constraint in Weckesser et al. [35] is the following: In ours, the lower- and upper-bound intervals are multiplied by the amount of parent feature clones (p_f). Our group instance constraint compares the sum of all children feature instances ($\sum_{c_f \in children(p_f)} c_f$) to the upper and lower bounds of the group instance interval ($\lambda_{u/l}^{GI}$) multiplied by the number of parent feature clones. In our running example (Figure 1), the parent feature *Software* and its group instance cardinality $\langle 1, 2 \rangle$ add a constraint like this: $1 * f_1 \leq f_2 + f_3 \leq 2 * f_1$.

$$\lambda_l^{GT}(p) * a_p \leq \sum_{c_f \in children(f_i)} t_{c_f} \leq \lambda_u^{GT}(p) * a_p \tag{10}$$

$$t_{c_f} = \begin{cases} 1 & , c_f \geq 1 \\ 0 & , c_f < 1 \end{cases} \tag{11}$$

$$a_{p_f} = \begin{cases} 1 & , p_f \geq 1 \\ 0 & , p_f < 1 \end{cases} \tag{12}$$

$$t2_{c_f} = \begin{cases} 1 & , c_f \geq p_f \\ 0 & , c_f < p_f \end{cases} \tag{13}$$

Group Type Cardinalities are represented using two different constraints. The first one (shown in Equation 10) conforms to the constraint in Weckesser et al. [35] for group type cardinality. We compare the sum of each child feature constant fed to the predicate (t_{c_f}) – which indicates whether the feature is greater than or equal to one (1) or not (0) (cf. Equation 11) – to the lower and upper group type cardinality bounds ($\lambda^{GT}_{u/l}$) multiplied to the predicate a_{p_f} – which represents the activity status of the parent feature (cf. Equation 12). For instance-generation operations, we need just the group-type cardinality constraint in Equation 10 because different valid configurations may support different features under a parent. In order to perform an upper bound analysis operation, we add another group type cardinality constraint. This constraint has the same structure as the one in Equation 10. However, we do change the predicate evaluating whether a feature is or is not active for the group type cardinality (t_{c_f}) to the predicate $t2_{c_f}$ represented in Equation 13. This definition ensures that the features are taken into account only if the number of instances of the feature is at least equal to the instances of the parent.

In our running example in Figure 1, for the group type cardinality of the feature *ComputingHardware*, the first-mentioned group type cardinality constraint is necessary to check if one of the child features is not active, thereby restricting the solver to only valid multi-set configurations. The second group type is necessary to find the false upper bound of feature *CPU*, which is not found by the first group type constraint alone. The reason is that with only the first group type constraint, the clones are not maximized with respect to all possible parent features.

The requires and exclude constraints of the multi-set approach follow the same behavior as those seen in the cloning approach, but with this important change: In the multi-set approach, it is not necessary to sum up all possible clones, because it suffices just to check with the given feature constant (cf. Equation 5 and Equation 6).

The constraints of multi-set make assumptions only about the overall number of feature clones. Consequently, with respect to our local interpretation of cardinalities, multi-set cannot find valid configurations, nor can it perform gap detection. For our purposes, therefore, multi-set may cause invalid configurations; basically, multi-set cannot be used to determine how many instances are under which clone.

Nonetheless, our reasons for pursuing investigations into multiset rest on two important characteristics of the approach. One, the bound analysis of multi-sets is much faster, as demonstrated in our evaluation (Section 5); and two, preprocessing with multiset bound analysis could possibly benefit even the cloning approaches.

4 Analysis Operations

In this section, we introduce the analysis operations that have to be encoded for automated reasoning with solvers, in order to define to the solver which anomalies are to be analyzed, with the given representation of the CFM from Section 3. The solver is called with the respective CFM solver representation to evaluate whether a CFM has a valid configuration. Thus, if the solver with the given input returns an unsatisfiability status, no valid configuration can be found, and the CFM is referred to as a **void CFM**.

For the analysis of whether a feature instance cardinality has gaps, which means a certain number of clones of the feature under one parent in the interval are not part of any valid configuration, cloning approaches are necessary. Consequently, to check if a feature has a **gap** at a cardinality *i*, the CFM solver representation of one of the cloning approaches is joined in a logical *and* by the sum of all clones of the feature under one parent asserted to the given cardinality *i* (cf. line 1 in Table 1). If the solver returns SAT, then there is a valid configuration with *i* clones of the feature under at least one parent clone, and thus there is no gap at the given cardinality. If the solver returns UNSAT, no valid configuration is found, and consequently, a gap is detected at the given cardinality of the feature.

With the given gap analysis, the cardinality of every feature instance of every feature in a CFM can be evaluated. Consequently, it can be expanded to analyze anomalies such as dead and false optional features, which are known from Boolean FM analysis. Thus, if a feature has a gap in every possible feature instance cardinality, it can also be considered dead. If the feature instance interval contains the number zero, but the gap analysis indicates a gap at cardinality zero, the feature can also be considered a false optional feature.

We introduce the bound analysis operation, which can be used to analyze the bounds of the given feature instance intervals. For the solver representation of this operation, we distinguish two cases. The first case employs the multi-set representation, which is combined through a logical *and* operation by the maximization or minimization operation on the given feature constant (cf. line 2 in Table 1). Accordingly, if the maximization of the feature constant returns a value less than the upper bound of the feature instance interval, a new upper bound is found. Additionally, if the maximization returns zero, the feature is considered dead, and if the minimization finds a new lower bound greater than zero, the feature is considered false-optional. In fact, both gap

Table 1. Solver representation of used anomaly analysis operations

$gap(f,i) := \neg SAT(CFM_{Cloning} \wedge ((\sum_{c_{cl} \in clones(f)} c_{cl}) = i))$
$bound_{max/min}(f) := SAT(CFM_{MS} \wedge (maximize/minimize f))$
$bound_{max/min}(f) := SAT(CFM_{Cloning} \wedge (maximize/minimize(\sum_{c_{cl} \in clones(f)} c_{cl})))$
$validConf(config) := SAT(CFM \wedge config)$

and bound analysis can also be used to detect the anomalies, dead, and false-optional features. The second case employs the cloning representation, which is also joined through a logical *and* operation by the maximization or minimization (cf. line 3 Table 1). The difference is that in the cloning approach, the sum of all feature instance variables under one parent is required, and not only the feature variable.

Finally, to evaluate whether a given configuration is valid in the CFM representation, the constants of the solver are mapped to the configuration's values, and if the solver returns SAT, the configuration is valid.

5 Evaluation

We evaluate the feasibility and performance of different approaches and their representation in three solvers using a set of 34 manually created, synthetic CFMs. While 13 CFMs contain anomalies, 21 are anomaly free. In the replication package [13] , we list the 34 subject systems and show important characteristics: depth of the tree (average: 3.6), number of features (average: 38), number of constraints (average: 1.6) and Big-M value (average: 60 981), which represents the maximum value of the multiplications of the maximum upper bounds of each branch in the feature tree. The Big-M value is used as a complexity measure as it combines the extremes of tree depth and interval sizes. The CFMs used in our evaluation are of a similar form as the ones used by Weckesser et al. [35] and Schnabel et al. [26], who generated Boolean FMs using BeTTy [27] and enriched them with cardinalities afterwards. However, as neither their data nor their generation tooling is available, we can not compare directly to their respective results. We thus chose to implement our own CFMs generator based on their description. Additionally, we counted the number of anomalies within each CFM. We documented the number of anomalies identified by the cloning representations and the multi-set representation. We evaluate the encodings to answer the following research questions:

RQ1 *How feasible are the proposed approaches for different CFM analyses?* We encoded the different representations into the three solvers and evaluated whether the encoding of each representation in each solver is feasible for creating valid configurations, finding the defined anomalies, and being encoded with less effort. Less effort in this case indicates whether the encoding can be created automatically and in a reasonable time frame. The results of this research question are presented in Section 5.1

RQ2 *How well do the different solvers perform on the different analysis operations with the different approaches?*

In the second research question, a comparison of the performance of all encodings across different solvers is made. For each analysis operation, solvers are called with every synthesized model and the different representations of the CFMs within the respective models, and the time required for this operation is tracked. Consequently, both (1) a comparison within each solver and (2) a comparison between the different solvers are conducted if the analysis operation is feasible in the respective representation, as evaluated in **RQ1**. For both parts of the evaluation, a Mac Mini 2023 with the Apple M2 Chip and 16 GB of RAM running macOS Sonoma 14.5 was used. During the experiment, we did not run other non-operating system software in parallel to reduce potential deviation in runtime measurements. The results of this research question are presented in Section 5.2.

5.1 RQ1: Feasibility of the different Representations

Regarding the feasibility of creating the different solver representations and using them to detect different analyses, we first consider whether the different solver representations can be created for all subjects CFMs and if the representation can be generated in a reasonable time frame. As discussed earlier, the ILP solver can only be used for optimization problems, and thus, only the multi-set solver representation is created. This is because the cloning approach is created for gap analysis, which is not an optimization problem. Therefore, only a bound analysis is conducted with the ILP solver, as this is an optimization problem. However, for the CSP and SMT solvers, the two cloning solver representations are generated. In summary, the creation of the various solver representations is considered viable. This is evidenced by the fact that the solver representations for all subjects were generated in less than one minute.

In the next step, using all the different solver representations, the anomaly analysis is conducted to determine whether they are feasible for finding all anomalies in the subjects. The various cloning solver representations identified all the anomalies through a conducted bound and gap analysis, which can be seen in the analysis table of the replication package [13]. Contrary to that, the multi-set representations were not feasible for identifying the gaps within the subjects. This is because the multi-set representations only consider the total number of feature instances and do not break it down for each parent instance.

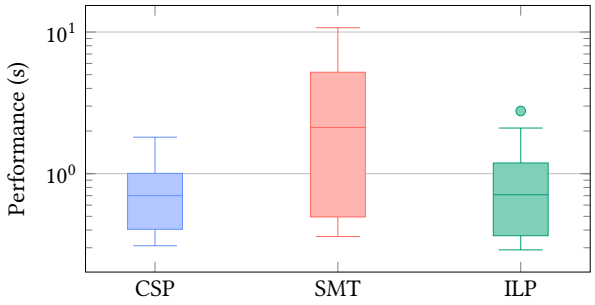

Figure 2. Performance comparison of multi-set bound analysis with three solver encodings (in seconds)

Our subject CFMs lack anomalies that are not at the bounds of the feature instance interval, so-called gaps, which is why for most of our subject CFMs, all solver representations detected all anomalies. For future work, we want to further extend the number of subject systems, especially in an industrial context, to conduct further analysis, investigate how often such gaps occur, and determine whether a bound analysis would be sufficient to identify most anomalies.

RQ1: Our analysis showed that across all representations and solvers, it is possible to identify false bounds and analyze whether a CFM is void. For gap analysis and checking valid configurations, only cloning representations with the CSP and SMT solvers can be used, as these tasks require a non-optimizing solver and local representations that preserve the tree structure of the CFMs.

5.2 RQ2: Anomaly Analysis Performance

After investigating the feasibility of the solver representation for certain analysis operations on the subject CFMs, we analyzed the performance of various representations across different solvers. We performed three experiments. First, we run the experiment using the bound analysis, then the gap analysis, and as a last step, consider a preprocessing step to reduce the effort of the gap analysis. For all analysis steps, we measure the time needed in seconds for each CFM.

Bound Analysis. In order to conduct the bound analysis, for each subject CFM, each solver is called with the respective analysis command and the encoded CFM as input. The time is recorded from the start of the call until the results are returned. Thus, the time includes loading the CFM from a JSON file, encoding it into the solver representation, and calling the bound analysis operation with the solver representation. The bound analysis is divided into maximize and minimize operations for each feature, resulting in two solver calls per feature. Figure 2 shows the boxplot of the multi-set bound analysis conducted with the three different solvers. The CSP and ILP solvers perform the bound analysis for most of the subject CFMs within one second. However, the SMT solver takes significantly longer for most of the CFMs.

We briefly consider now some of the subject CFMs. CFMs are considered big if they have many features and a large Big-M. For example, big CFMs include *model_6* with 62 features, a Big-M value of 70 520, and *model_11* with 78 features and a Big-M value of 112 500. For these big CFMs, the CSP solver needed 1.3s and 1.8s, the ILP solver needed 1.6s and 2.7s, and the SMT solver needed 8.8s and 10.73s. By contrast, all solvers demonstrate faster performance (i.e., response time of less than one second) on small CFMs like *multiplayer*, *pizzas*, *sandwich*, and *soccer*, which have 5-15 features and Big-M between 8 and 450. The subject systems, *model_5* and *model_8*, have the same number of features but differ in their Big-M values, i.e., 35 features and Big-M values of 43 776 and 80 370, respectively. The bound analysis with the multi-set approach on the three different solvers showed little impact on the nearly doubled number of Big-M, with analysis times like 0.88s and 0.62s for CSP, 0.72s and 0.56s for ILP, and 2.00s and 2.12s for SMT. We found similar results for subject systems with an overall higher but equal number of features, i.e., *model_0*, *model_9*, and *model_18* have 67 features but Big-M values of 92 500, 88 200, and 108 288, respectively. Lastly, subject systems with similar Big-M value but a variable number of features did not show a significant difference in analysis time. For instance, *model_0* (67 features and Big-M of 92 500), *model_1* (59 features and Big-M of 92 250), and *model_27* (70 features and Big-M of 92 400) have similar analysis time across the solvers, i.e., 1.52s, 1.26 and 0.63s for CSP, 7.8s, 6.99s, and 2.01s for ILP, and 2.10s, 0.93s and 0.84s for SMT. In a different example, *model_10*, *model_11* and *model_16* consist of 65, 78, and 34 features, but have similar Big-M value with 112 800, 112 500, and 112 700, respectively. For these models, the solver analysis time shows some variance as CSP needed 1.21s, 1.81s, and 0.61s, ILP needed 6.85s, 10.73s, and 3.98s, and SMT needed 1.36s, 2.77s, and 0.55s.

Following the evaluation of the three multi-set solver representations, we consider the cloning solver representation to conduct a comprehensive bound analysis, ensuring the thoroughness of the evaluation process. As the cloning solver representations, by definition, have a much larger set of constants and consequently a much larger encoding size, the solver performance is also negatively affected. This can be demonstrated by looking at *model_12* with 51 features and a Big-M of 81 900, which needs 0.99s in the CSP multi-set solver representation and the same CFM in the best cloning bound analysis solver representation, namely CSP cloning with integer leaves, needs 144 115.55s (approx. 40 hours). Also, when considering smaller CFMs like *multiplayer* with 15 features and a Big-M of 8, where the best multi-set solver representation needed 0.39s, the best cloning solver representation needed 0.7s, which is much closer than the example before due to the fact that small CFMs also have a smaller impact on the size of the cloning encoding because a much smaller number of constants are needed.

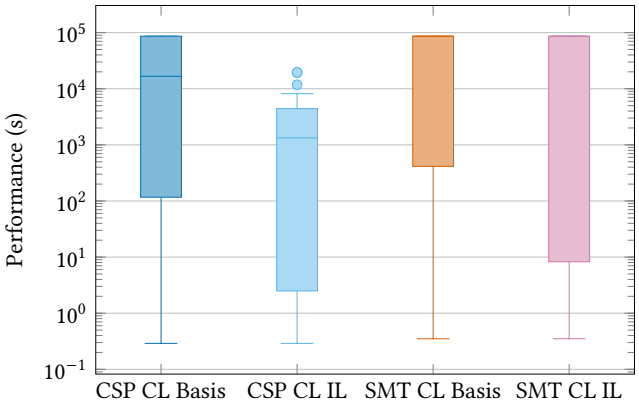

Figure 3. Performance comparison of gap analysis

RQ2 Bound Analysis: Our analysis showed that the best solver representation for performing a bound analysis is the CSP multi-set representation, which significantly outperformed both the cloning approaches and the SMT multi-set representation. On the other hand, the ILP multi-set solver representation was very close to the performance of the CSP multi-set representation.

Gap Analysis. As delineated in the feasibility analysis, the gap analysis exclusively encompasses the two cloning solver representations, in addition to the CSP and SMT solvers. Due to time constraints for this evaluation, we limited each gap analysis to 24 hours per model. The boxplot in Figure 3, which represents the overall performance of the different solver representations, shows out that only one of the encodings could execute the gap analysis for all models within the time limit. Consequently, the CSP cloning with integer leaves solver representation is by far the most performant solver representation for gap analysis.

It is noteworthy that even if both SMT cloning solver representations could not perform the whole analysis on all subject CFMs, the boxplot represents the impact of the integer leaves on the performance of the analysis compared to the cloning basis approach. For example, considering the smaller CFMs (cf. *multiplayer*, *pizza* and *sandwich*) with 6-15 features and a Big-M value between 1 and 30, all solvers and encodings needed close around one second to perform the full gap analysis, and consequently, no big difference between the two representations is measured. Conversely, for a considerable number of features and a substantial Big-M value (cf. *model_11* with 78 features and a Big-M of 112 500), the two SMT cloning solver representations exceeded the time limit, while both CSP representations successfully completed the task within the specified timeframe. When having a closer look at the time needed for both CSP representations, the basis cloning representation took 68 364s and the updated representation with integer leaves only 4568s, which underlines the impact of the integer leaves representation.

Considering different models with a similar number of features but differing Big-M values, i.e., *model_0*, *model_9*, and *model_18*, all having 67 features but differing Big-M values of 92 500, 88 200, and 108 288, no significant influence from the Big-M values can be perceived. For example, when looking at the gap analysis with the CSP integer leaves representation, *model_18*, with the highest Big-M Value, needed the shortest time 7281.42s, compared to *model_9* with 19 522.86s and *model_0* with 8164.65s. On the other hand, when comparing models like *model_10*, *model_11*, and *model_16*, all having Big-M values around 112 000 (112 800, 112 500, 112 700) but differing number of features 65, 78, and 34, no significant impact is apparent. For example, considering gap analysis with the CSP integer leaves approach, the three models needed 19 440.22s, 11 765.54s, and 1776.72s, which shows that *model_10* with 65 features needed more time than *model_11* with 78 features. In contrast, *model_16*, with the fewest features, also required the least time.

We briefly consider some of the CFMs again. For subject systems with similar Big-M value but a variable number of features, we found that only CSP and CSP integer leaves representation computed results within our time limit. Interestingly, while computation time generally increased with the number of features as expected, there were interesting outliers as well. Consider *model_8* with 35 features and a Big-M value of 80 370, *model_25* with 50 features and a Big-M value of 80 688, and *model_12* with 51 features and a Big-M value of 81 900. CSP needed 24 068.86s for *model_8*, 76 332.25s for *model_25*, and 68 364.86s for *model_12* to compute a result. However, the CSP integer leaves representation needed 1704.46s, 4316.40s, and 4568.43s, respectively. While there is a major speed-up from the CSP basis to the CSP integer leaves representation, it is interesting that CSP took less time for the more complex model in *model_12* than for *model_25*.

In a second step, we included a pre-processing step, in which a bound analysis was first conducted with the multi-set solver representation, and the detected false bounds were updated in the subject CFMs. Then we conducted the gap analysis as explained before and compared it to the performance measured without these pre-processing steps. Of course, the CFMs without false bounds has no impact on the performance because the models did not change. But even in the model, where false bounds were detected and updated, the analysis showed that the pre-processing step only slightly reduced the needed time, for example, for CFM *model_0* with 67 features and a Big-M value of 92 500 from 8164.65s to 8148.64s. Also, for smaller CFMs like *multiplayer* with 15 features and a Big-M value of 8, the needed time was reduced from 0.67s to 0.46s. These results are restricted to our synthesized models and the number and size of false bounds, and therefore should be further explored with larger models that include more anomalies.

RQ2 Gap Analysis: Our analysis showed that the best solver representation for performing a gap analysis is the CSP cloning with integer leaves representation, which significantly outperformed the other cloning solver representations. Also, the analysis showed that a pre-processing bound analysis step has little impact on the overall gap analysis performance.

5.3 Threats to Validity

The generalizability of our evaluation may be limited by the synthesized CFMs we used. To mitigate this threat, we ensured that no author of this paper was involved in the creation of the CFMs. The CFMs used are of a similar form to those used in literature to evaluate related approaches [26, 35]. The synthetic nature also makes it easier to control FM size, feature count, and Big-M to evaluate the scalability of the encodings and solvers. However, the solver comparison could benefit from further enhancements by evaluating CFMs in an industrial context. Furthermore, only one solver per category is used in our analyses. These solvers are, to the best of our knowledge, current state-of-the-art. Nevertheless, we believe the results allow a good comparison between the solver types, even if specific instances may differ in performance. We expect an investigation of other solvers in the stated categories (such as different SMT, CSP, or ILP solvers) and solvers with optimizations (e.g., Z3 [12] with its Z3-specific tactics) to yield similar results. Our different encodings are not formally proven to be correct, as we only evaluated them on a limited set of CFMs and compared results across encodings, solvers, and approaches.

6 Related Work

A number of studies have been conducted on the automated analysis of CFMs with a global interpretation of the cardinalities [26, 35]. This analysis is less complex than with the local interpretation. In these works, a ILP solver is employed for the purpose of bound analysis, while an SMT solver is utilized for gap analysis. A notable distinction between the two gap analyses is the global interpretation employed in the former, which diverges from the local interpretation utilized in our work. As previously elucidated, the aforementioned approach is employed in our work as the multi-set approach. Furthermore, Bontemps et al. [6] extend the FODA FM with certain cardinalities like relative cardinalities, but do not exactly match all cardinality extensions of our approach [20, 28]. In these works, the authors approached the basis cloning representation used in our work, which we extended to our needs. As a solver the authors used the Choco CSP Solver [25]. In addition to the related work dealing with CFMs or similar FMs with additional cardinality constraints, we also looked at other analysis approaches where cardinality plays an important role, such as UML models and *CLAFER* models [7, 8, 19, 33, 34]. In the works for analyzing the *CLAFER* models [7, 33, 35], the authors also used a multi-set semantics similar to the approach for CFMs by Schnabel et al. [26]. Also, for Unified Modeling Language (UML) models, different authors approached different reasoning methods for the finite satisfiability of UML models [8, 19]. These methods can not be directly mapped to the problems considering our cardinality constraints, but provide a general understanding of the concept of mapping problems or diagram restriction, like hierarchies with generalization sets [19], into constraints for a CSP solver.

7 Conclusion

In this paper, we compared different representations (multi-set, cloning basis, and cloning with integer leaves) for the analysis of CFMs utilizing different solver categories (CSP, ILP, and SMT). Our feasibility analysis of the solver representations showed that the cloning representations are feasible to find all anomalies by utilizing the two solver categories CSP and SMT. Contrary, the multi-set representation and the ILP solver are only feasible to analyse the bounds of the cardinality intervals. In light of the observed performance, it can be concluded that the CSP solver category demonstrated superior performance in comparison to the other two solver categories with all different representations. With respect to the different representations, the cloning with integer leaves exhibited superior performance in comparison to the cloning basis variant for the gap analysis. Furthermore, the multi-set representation demonstrated the best performance in the context of only bound analysis. Additionally, we could not notice a direct impact of the two properties of CFMs (Big-M value and the number of features) on the analysis time, when comparing subject systems having close values for one property but differing for the other.

In subsequent studies, we intend to broaden the scope of the evaluation by encompassing additional models, particularly those drawn from industrial contexts that are larger in size and more complex overall. In addition, the objective of this study is to investigate the implications of unbounded CFMs. Moreover, the objective is to broaden the scope of analysis operations in relation to existing FM analysis. This expansion entails the consideration of supplementary analysis operations, including the identification of core features and core cardinalities. Furthermore, the objective is to enhance the existing analysis to facilitate the explanation of the observed anomalies and the influence of CFM dimensions on performance.

Acknowledgments

This work was funded by the Deutsche Forschungsgemeinschaft (DFG, German Research Foundation) – CRC 1608 – 501798263, SCHA1635/15-1. This paper has been edited by our textician, Daniel Shea.

References

[1] Sven Apel, Don Batory, Christian Kästner, and Gunter Saake. 2013. *Software Product Lines*. Springer Berlin Heidelberg, Berlin, Heidelberg, 3–15. doi:10.1007/978-3-642-37521-7_1

[2] Clark Barrett, Pascal Fontaine, and Cesare Tinelli. 2016. The Satisfiability Modulo Theories Library (SMT-LIB). `www.SMT-LIB.org`.

[3] David Benavides, Sergio Segura, and Antonio Ruiz-Cortés. 2010. Automated analysis of feature models 20 years later: A literature review. *Information Systems* 35 (Sept. 2010), 615–636. doi:10.1016/j.is.2010.01.001

[4] Thorsten Berger, Ralf Rublack, Divya Nair, Joanne M. Atlee, Martin Becker, Krzysztof Czarnecki, and Andrzej Wąsowski. 2013. A survey of variability modeling in industrial practice. In *Proceedings of the 7th International Workshop on Variability Modelling of Software-Intensive Systems* (Pisa, Italy) *(VaMoS '13)*. Association for Computing Machinery, New York, NY, USA, Article 7, 8 pages. doi:10.1145/2430502.2430513

[5] A. Biere, M. Heule, and H. van Maaren. 2009. *Handbook of Satisfiability*. IOS Press. https://books.google.at/books?id=shLvAgAAQBAJ

[6] Yves Bontemps, Patrick Heymans, Pierre-Yves Schobbens, and Jean-Christophe Trigaux. 2004. Semantics of FODA feature diagrams. In *Proceedings SPLC 2004 Workshop on Software Variability Management for Product Derivation–Towards Tool Support*. 48–58.

[7] Kacper Bąk, Zinovy Diskin, Michał Antkiewicz, Krzysztof Czarnecki, and Andrzej Wąsowski. 2016. Clafer: unifying class and feature modeling. *Software & Systems Modeling* 15, 3 (July 2016), 811–845. doi:10.1007/s10270-014-0441-1

[8] Marco Cadoli, Diego Calvanese, Giuseppe De Giacomo, and Toni Mancini. 2007. Finite Model Reasoning on UML Class Diagrams Via Constraint Programming. In *AI*IA 2007: Artificial Intelligence and Human-Oriented Computing*, Roberto Basili and Maria Teresa Pazienza (Eds.). Springer, Berlin, Heidelberg, 36–47. doi:10.1007/978-3-540-74782-6_5

[9] P. Clements and L. Northrop. 2002. *Software Product Lines: Practices and Patterns*. Addison-Wesley.

[10] Krzysztof Czarnecki, Simon Helsen, and Ulrich Eisenecker. 2005. Formalizing cardinality-based feature models and their specialization. *Software Process: Improvement and Practice* 10, 1 (2005), 7–29. arXiv:https://onlinelibrary.wiley.com/doi/pdf/10.1002/spip.213 doi:10.1002/spip.213

[11] Adnan Darwiche and Pierre Marquis. 2002. A knowledge compilation map. *J. Artif. Int. Res.* 17, 1 (sep 2002), 229–264. https://doi.org/10.1613/jair.989

[12] Leonardo de Moura and Nikolaj Bjørner. 2008. Z3: An Efficient SMT Solver. In *Tools and Algorithms for the Construction and Analysis of Systems*, C. R. Ramakrishnan and Jakob Rehof (Eds.). Springer, Berlin, Heidelberg, 337–340. doi:10.1007/978-3-540-78800-3_24

[13] Fabian Eger, Lukas Güthing, and Kevin Feichtinger. 2026. *KIT-TVA/cfmtoolbox-solver-representations: Replication Package*. doi:10.5281/zenodo.20080331

[14] Alexander Felfernig, Andreas Falkner, and David Benavides. 2024. *Feature Models: AI-Driven Design, Analysis and Applications*. Springer International Publishing, Cham. doi:10.1007/978-3-031-61874-1

[15] Google. 2024. CP-SAT Solver OR-Tools. https://developers.google.com/optimization/cp/cp_solver

[16] Google. 2024. Linear Optimization | OR-Tools | Google for Developers. https://developers.google.com/optimization/lp

[17] Lukas Güthing, Mathis Weiß, Ina Schaefer, and Malte Lochau. 2024. Sampling Cardinality-Based Feature Models. In *Proceedings of the 18th International Working Conference on Variability Modelling of Software-Intensive Systems* (Bern, Switzerland) *(VaMoS '24)*. Association for Computing Machinery, New York, NY, USA, 46–55. doi:10.1145/3634713.3634719

[18] Kyo C Kang, Sholom G Cohen, James A Hess, William E Novak, and A Spencer Peterson. 1990. *Feature-oriented domain analysis (FODA) feasibility study*. Technical Report. Carnegie-Mellon Univ., Pittsburgh, Pa, Software Engineering Inst. doi:10.21236/ADA235785

[19] Azzam Maraee and Mira Balaban. 2007. Efficient reasoning about finite satisfiability of UML class diagrams with constrained generalization sets. In *Model Driven Architecture-Foundations and Applications: Third European Conference, ECMDA-FA 2007, Haifa, Israel, June 11-15, 2007, Proccedings 3*. Springer, 17–31. https://doi.org/10.1007/978-3-540-72901-3_2

[20] Raúl Mazo, Camille Salinesi, Daniel Diaz, and Alberto Lora-Michiels. 2011. Transforming attribute and clone-enabled feature models into constraint programs over finite domains. In *6th International Conference on Evaluation of Novel Approaches to Software Engineering (ENASE)*. https://hal.science/hal-00707546

[21] Jens Meinicke, Thomas Thüm, Reimar Schröter, Fabian Benduhn, Thomas Leich, and Gunter Saake. 2017. *Mastering Software Variability with FeatureIDE*. Springer. doi:10.1007/978-3-319-61443-4

[22] Raphael Michel, Andreas Classen, Arnaud Hubaux, and Quentin Boucher. 2011. A formal semantics for feature cardinalities in feature diagrams. In *Proceedings of the 5th International Workshop on Variability Modeling of Software-Intensive Systems* (Namur, Belgium) *(VaMoS '11)*. Association for Computing Machinery, New York, NY, USA, 82–89. doi:10.1145/1944892.1944902

[23] Klaus Pohl, Günter Böckle, and Frank J van der Linden. 2005. *Software Product Line Engineering: Foundations, Principles and Techniques*. Springer Science & Business Media. doi:10.1007/3-540-28901-1

[24] Matias Pol'la, Agustina Buccella, and Alejandra Cechich. 2021. Analysis of variability models: a systematic literature review. *Software and Systems Modeling* 20, 4 (01 Aug 2021), 1043–1077. doi:10.1007/s10270-020-00839-w

[25] Charles Prud'homme and Jean-Guillaume Fages. 2022. Choco-solver. *Journal of Open Source Software* 7, 78 (2022), 4708. https://hal.science/hal-03932507v1

[26] Thomas Schnabel, Markus Weckesser, Roland Kluge, Malte Lochau, and Andy Schürr. 2016. CardyGAn: Tool Support for Cardinality-based Feature Models. In *Proceedings of the 10th International Workshop on Variability Modelling of Software-Intensive Systems* (Salvador, Brazil) *(VaMoS '16)*. Association for Computing Machinery, New York, NY, USA, 33–40. doi:10.1145/2866614.2866619

[27] Sergio Segura, José A. Galindo, David Benavides, José A. Parejo, and Antonio Ruiz-Cortés. 2012. BeTTy: benchmarking and testing on the automated analysis of feature models. In *Proceedings of the 6th International Workshop on Variability Modeling of Software-Intensive Systems* (Leipzig, Germany) *(VaMoS '12)*. Association for Computing Machinery, New York, NY, USA, 63–71. doi:10.1145/2110147.2110155

[28] Gustavo Sousa, Walter Rudametkin, and Laurence Duchien. 2016. Extending feature models with relative cardinalities. In *Proceedings of the 20th International Systems and Software Product Line Conference (SPLC '16)*. Association for Computing Machinery, New York, NY, USA, 79–88. doi:10.1145/2934466.2934475

[29] Peter J Stuckey, Thibaut Feydy, Andreas Schutt, Guido Tack, and Julien Fischer. 2014. The minizinc challenge 2008–2013. *AI magazine* 35, 2 (2014), 55–60. https://www.minizinc.org/challenge/

[30] Chico Sundermann, Tobias Heß, Michael Nieke, Paul Maximilian Bittner, Jeffrey M. Young, Thomas Thüm, and Ina Schaefer. 2023. Evaluating state-of-the-art # SAT solvers on industrial configuration spaces. *Empirical Software Engineering* 28, 2 (13 Jan 2023), 29. doi:10.1007/s10664-022-10265-9

[31] Thomas Thüm, Sven Apel, Christian Kästner, Ina Schaefer, and Gunter Saake. 2014. A Classification and Survey of Analysis Strategies for Software Product Lines. *ACM Comput. Surv.* 47, 1, Article 6 (June 2014), 45 pages. doi:10.1145/2580950

[32] Edward Tsang. 1999. A glimpse of constraint satisfaction. *Artificial Intelligence Review* 13 (1999), 215–227. doi:10.1023/A:1006558104682

[33] Markus Weckesser. 2018. Mathematical Programming for Anomaly Analysis of Clafer Models | Proceedings of the 21th ACM/IEEE International Conference on Model Driven Engineering Languages and Systems. https://dl.acm.org/doi/10.1145/3239372.3239398

[34] Markus Weckesser, Malte Lochau, Michael Ries, and Andy Schürr. 2017. Towards complete consistency checks of Clafer models. In *Proceedings of the 8th ACM SIGPLAN International Workshop on Feature-Oriented Software Development (FOSD 2017)*. Association for Computing Machinery, New York, NY, USA, 11–20. doi:10.1145/3141848.3141850

[35] Markus Weckesser, Malte Lochau, Thomas Schnabel, Björn Richerzhagen, and Andy Schürr. 2016. Mind the Gap! Automated Anomaly Detection for Potentially Unbounded Cardinality-Based Feature Models. In *Proceedings of the 19th International Conference on Fundamental Approaches to Software Engineering - Volume 9633*. Springer-Verlag, Berlin, Heidelberg, 158–175. doi:10.1007/978-3-662-49665-7_10

Received 2026-03-04; accepted 2026-04-23