The Journal of Systems and Software 227 (2025) 112434

Contents lists available at ScienceDirect

The Journal of Systems & Software

journal homepage: www.elsevier.com/locate/jss

FMSans: An efficient approach for constraints removal and parallel analysis of feature models ☆

Jose-Miguel Horcas *, Joaquín Ballesteros, Mónica Pinto, Lidia Fuentes

ITIS Software, Universidad de Málaga, Andalucía Tech, Bulevar Louis Pasteur 35, 29010, Malaga, Spain

ARTICLE INFO

Keywords:
Automated analysis
Constraint
Feature model
Feature tree
Parallelization
Software product line

ABSTRACT

Cross-tree constraints help to compact feature models by using arbitrary propositional logic formulas, which efficiently capture interdependencies between features. However, the existence of these constraints increases the complexity of reasoning about feature models, whether we use SAT solvers or compile the model to a binary decision diagram for efficient analyses. Although some works have tried to refactor constraints to eliminate them, they deal only with simple constraints (i.e., requires and excludes) or require introducing an additional set of features, increasing the size and complexity of the resulting feature model. This paper presents an approach that eliminates all the cross-tree constraints in regular boolean feature models, including arbitrary constraints in propositional logic formulas. Our approach for removing constraints consists of splitting the semantics of feature models into orthogonal disjoint feature subtrees, which are then analyzed in parallel to alleviate the exponential blow-up in memory of the resulting feature tree. We propose a codification of the constraints and define and analyze different heuristics for constraints ordering to reduce the complexity of identifying the valid disjoint subtrees when removing constraints.

1. Introduction

Feature models (Kang et al., 1990) are widely used to express the variability of *software product lines* (SPL) (Apel et al., 2013). A feature model consists of a *feature tree* and a set of constraints. In order to reason about feature models they are translated into some formalization (Benavides et al., 2010), and then some solver is applied. The most common options are to express feature models in propositional logic and use SAT solvers or constraint programming with CSP solvers. Alternatively, they can be mapped to other data structures (a.k.a. knowledge compilation (Darwiche and Marquis, 2002)), such as binary decision diagrams (BDD) (Mendonça et al., 2008; Heradio et al., 2016). Although the translation of the feature model to SAT or CSP can be performed in polynomial time, the analysis of the resulting program is NP-complete (Mendonça et al., 2009). On the contrary, using BDDs, it is possible to reduce the analysis complexity regarding time in a more tractable problem (e.g., checking whether a feature model has products or not is performed in constant time using a BDD solver) (Mendonça, 2009). However, the construction of the BDD can be intractable in memory and time in most cases (Heß et al., 2021; Thüm, 2020).

The difficulty of analyzing feature models is mainly due to *cross-tree constraints* (*constraints* for short), rather than to the feature tree complexity and size. A feature tree is a hierarchically organized tree structure that decomposes the features of a system into or-groups, alternative groups, mandatory and optional features, and possibly other kinds of relationships (Schobbens et al., 2007). A constraint expresses an interaction of features not captured by the feature tree, commonly represented as a textual logical formula. Two types of constraints are distinguishable: (1) *simple constraints* (Kang et al., 1990; Schobbens et al., 2007), which include *requires constraints* (i.e., the presence of feature f_1 in a product implies the presence of feature f_2) and *excludes constraints* (i.e., two features are mutually exclusive and cannot be present together in a product); and (2) *complex constraints* (Batory, 2005; Knüppel et al., 2017) which are arbitrary propositional logic formulas over the set of features. Note that, the greater of the complexity of the constraints, the greater of the complexity of the analysis.

Indeed, the automated analysis of feature models (AAFM) (Benavides et al., 2010) without such constraints is easy, as simple recursive functions on trees can be applied (van den Broek and Galvão, 2009) without needing to translate the feature model into other formalization (e.g., SAT, CSP) or data structures (e.g., BDD). Therefore, one sound approach to simplify the AAFM consist of manipulating the feature model to eliminate constraints (Knüppel et al., 2017; Gil et al., 2010; van den Broek et al., 2008). However, the SPL community has put little effort into eliminating constraints from feature models, mainly due to

☆ Editor: Raffaela Mirandola.

* Corresponding author.
E-mail addresses: horcas@uma.es (J.-M. Horcas), jballesteros@uma.es (J. Ballesteros), mpinto@uma.es (M. Pinto), lfuentes@uma.es (L. Fuentes).

https://doi.org/10.1016/j.jss.2025.112434
Received 20 February 2024; Received in revised form 23 September 2024; Accepted 10 March 2025
Available online 10 April 2025
0164-1212/© 2025 The Authors. Published by Elsevier Inc. This is an open access article under the CC BY license (http://creativecommons.org/licenses/by/4.0/).

two reasons: (1) constraints play a crucial role in compacting feature models (Knüppel et al., 2017), and (2) the high cost of eliminating constraints (Gil et al., 2010). Gil et al. (2010) considered translations of any logical formula into a feature tree and concluded that constraints are inevitable and cannot be removed without introducing a new set of features. Knüppel et al. (2017) developed an algorithm to eliminate complex constraints by introducing new features and augmenting the number of simple constraints. van den Broek et al. (2008) propose an algorithm to eliminate simple constraints, delivering a feature tree where features may have multiple occurrences, a.k.a. *generalized feature tree* (GFT). The advantage of GFT is that it does not require translating the feature tree to SAT or BDD to reason about its properties (van den Broek and Galvão, 2009). However, the size of the resulting GFT grows exponentially in the number of constraints in the worst case. Moreover, Broek's algorithm (van den Broek et al., 2008) is not fully compatible with Knüppel's approach (Knüppel et al., 2017) because the new features introduced due to the removal of complex constraints are also part of simple constraints, and when removing these simple constraints, the semantics of the GFT may be altered (see Section 2).

In this paper, we propose an approach that eliminates all constraints, both complex and simple, of a feature model while maintaining its semantics and without introducing new features into the resulting feature tree. Our approach exploits parallel computing to remove the constraints and for the posterior analysis of the feature tree based on splitting the semantics of the feature model into orthogonal subtrees with disjoint semantics (van den Broek and Galvão, 2009). We make the following contributions:

- We formally present an approach that eliminates all constraints from a feature model (Section 3).
- We codify the constraints and define and analyze different heuristics for constraints ordering that can be applied when removing the constraints in order to reduce the complexity of identifying valid subtrees (Section 4).
- We translate the original feature model to a new representation of disjoint subtrees (called FM_{Sans}) by exploiting parallelism techniques to avoid the exponential blow-up in memory during the construction and analysis of the GFT (Section 5).
- We perform a parallel analysis of the subtrees, being able to perform analysis operations without using external solvers (Section 6).
- We evaluate our approach by providing a Python implementation of our proposal (Section 7), which can be integrated within existing AAFM libraries.
- We compare our work with the existing approaches that address the elimination of constraints in feature models (Table 7 in Section 8).

As far as we know, our approach is the first work that eliminates all constraints, including complex and simple constraints, for feature models with hundreds of constraints. With this contribution, we conjecture that our approach will open a new research window where AAFM can benefit from parallel computation. For instance, our FM_{Sans} model could be used with SAT solvers to perform analysis operations in parallel and with BDDs to alleviate the scalability problem in memory when constructing the BDD. This is especially helpful in domains such as mobile networks nowadays, where the network functions present high variability and can be modeled and configured in parallel (Gramaglia et al., 2022; Camelo et al., 2022).

This paper extends the work published as a conference paper in SPLC'23 (Horcas et al., 2023a) by adding the following contributions: (1) three new theorems with their respective proofs on which our approach relies (Section 3); (2) a new step in the FM_{Sans} approach to consider the ordering of the constraints when removing them (Section 3); (3) a set of four heuristics to analyze the ordering of the constraints in the codification for their removal (Section 4.2); (4) an analysis evaluation of how different orderings affect the performance of identifying the valid disjoint subtrees when building the FM_{Sans} model (Section 7); and (5) a new parallelizable analysis operation that can be executed directly over the feature tree: the generation of all products given a feature tree (Section 6).

2. Feature trees and constraints

This section introduces the different types of feature models, feature trees, and constraints we consider in this paper.

2.1. Generic feature models

Inspired by the formal semantics of Schobbens et al. (2007), we define a feature model as follows.

Definition 1 (*Feature Model*). A *feature model* (FM) is a tuple $(\mathcal{T}, \Psi)$ where $\mathcal{T}$ is the feature tree and Ψ is a set of constraints. ■

Definition 2 (*Feature Tree*). The feature tree $\mathcal{T}$ is a 5-tuple $(\mathcal{F}, \mathcal{F}_c, \mathcal{F}_a, r, \mathcal{R})$:

- $\mathcal{F}$ is a finite set of features. A *feature* $f \in \mathcal{F}$ is a characteristic or end-user-visible behavior of a software system (Apel et al., 2013).
- $\mathcal{F}_c \subseteq \mathcal{F}$ is a subset of concrete features. A *concrete feature* $f \in \mathcal{F}_c$ is a feature mapped to an implementation artifact (Thüm et al., 2014a).
- $\mathcal{F}_a \subseteq \mathcal{F}$ is a subset of abstract features. An *abstract feature* $f \in \mathcal{F}_a$ is a feature used for grouping and decomposing the feature tree.
- $r \in F$ is the *root feature*.
- $\mathcal{R} \subseteq \mathcal{F} \times \mathcal{F}^n \times \mathbb{N}^2$ is the finite set of decompositional relations between the features that hierarchically organize the tree structure of $\mathcal{T}$. Each relation $r \in \mathcal{R}$ is denoted as $r = \left(f, [g_1, g_2, \ldots, g_n], \langle a..b \rangle\right)$ meaning that f is the parent feature of subfeatures $g_i, 1 \leq i \leq n$, with a multiplicity $\langle a..b \rangle$ (Czarnecki et al., 2005). Whenever f is included in a configuration, at least a and at most b of the g_i's must also be included. In this way, the features are decomposed into either *or-groups* ($\langle 1..n \rangle$ and $n > 1$), *alternative-groups* a.k.a. *xor-groups* ($\langle 1..1 \rangle$ and $n > 1$), *optional features* ($\langle 0..1 \rangle$ and $n = 1$), and *mandatory features* ($\langle 1..1 \rangle$ and $n = 1$). ■

The feature tree $\mathcal{T}$ may be empty (called NIL), representing the empty feature model with no features and no products. Note that our formalization of $\mathcal{T}$ also supports additional decomposition relations such as *and-groups* (Schobbens et al., 2007) (i.e., a feature f can appear in more than one relation $r \in \mathcal{R}$ as a parent feature), custom *group cardinalities* (Czarnecki et al., 2005) ($\langle a..b \rangle$ with $a \geq 1$ and $b \leq n$ when $n > 1$), *mutex groups* (Berger et al., 2014) ($\langle 0..1 \rangle$ and $n > 1$), and *multiple decomposition types* for a feature (e.g., an alternative- and an or-group below the same feature). However, these additional decomposition relations have been demonstrated that can be refactored to the most basic relations (or-group, xor-group, optional, and mandatory features) (Knüppel et al., 2017; Horcas et al., 2023c). Thus, without loss of generality, in this paper, we assume that $\mathcal{T}$ contains only those basic relations and that a feature $f \in \mathcal{F}$ can appear in more than one relation $r \in \mathcal{R}$ as a parent feature only for mandatory and optional features (i.e., and-groups). We define the following functions to easily write concise algorithms:

`parent`: $\mathcal{F} \to \mathcal{F} \cup \{NIL\}$. Returns the parent of $f \in \mathcal{F}$ or NIL if f is the root.

`relation`: $\mathcal{F} \times \mathcal{F} \to \mathcal{R}$. Given two features $f, g \in \mathcal{F}$ returns the relation of f as a parent containing g as a child. A relation $r \in \mathcal{R}$ has four properties:

`parent`: $\mathcal{F}$. The parent of the relation, which is f.
`children`: $\mathcal{F}^n$. Set of subfeatures.
`card_min`: $\mathbb{N}$. Minimum cardinality (multiplicity).
`card_max`: $\mathbb{N}$. Maximum cardinality (multiplicity).

`is_optional`: $\mathcal{F} \to \mathbb{B}$. Returns true if feature $f \in \mathcal{F}$ is optional.

`is_mandatory`: $\mathcal{F} \to \mathbb{B}$. Returns true if feature $f \in \mathcal{F}$ is mandatory.

`is_or_group`: $\mathcal{F} \to \mathbb{B}$. Returns true if feature $f \in \mathcal{F}$ is an or-group feature.

`is_xor_group`: $\mathcal{F} \to \mathbb{B}$. Returns true if feature $f \in \mathcal{F}$ is an alternative group feature.

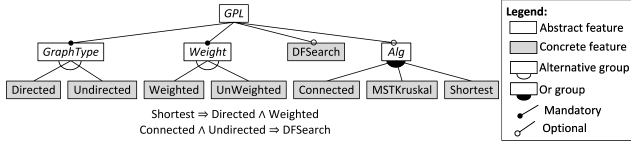


Fig. 1. Feature model excerpt of the Graph Product Line.

is_group: $\mathcal{F} \rightarrow \mathbb{B}$. Returns true if $f \in \mathcal{F}$ is an or-group or a xor-group.
is_leaf: $\mathcal{F} \rightarrow \mathbb{B}$. Returns true if feature $f \in \mathcal{F}$ has no children.

Definition 3 (*Constraints*)**.** The *constraints* (Ψ) is a set of additional textual constraints defined as arbitrary propositional formulas over the set of features $\mathcal{F}$, that is, $\Psi \subseteq \mathbb{B}(\mathcal{F})$. We split Ψ into *simple constraints* Ψ_s and *complex constraints* Ψ_c ($\Psi = \Psi_s \cup \Psi_c$).

- **Simple constraints.** Ψ_s is the set of simple constraints that includes the *requires* and *excludes* constraints ($\Psi_s = \Psi_{s_R} \cup \Psi_{s_E}$):
  - $\Psi_{s_R} \subseteq \{f \rightarrow g | f, g \in \mathcal{F}\}$ is the set of *requires constraints* where the constraints are in the form "f REQUIRES g".
  - $\Psi_{s_E} \subseteq \{f \rightarrow \neg g | f, g \in \mathcal{F}\}$ is the set of *excludes constraints* where the constraints are in the form "f EXCLUDES g".
- **Complex constraints.** Ψ_c is the set of complex constraints that includes the pseudo-complex and strict-complex constraints ($\Psi_c = \Psi_{c_P} \cup \Psi_{c_S}$):
  - Ψ_{c_P} is the set of *pseudo-complex constraints* whose constraints in conjunctive normal form (CNF) have the form $\bigwedge c_i$ where $c_i \equiv (\neg f_1 \vee f_2)$ or $c_i \equiv (\neg f_1 \vee \neg f_2)$ for arbitrary features $f1, f2 \in \mathcal{F}$. That is, pseudo-complex constraints are convertible to a set of simple constraints (*requires* and *excludes*) (Knüppel et al., 2017).
  - Ψ_{c_S} is the set of *strict-complex constraints* whose constraints in CNF are neither pseudo-complex nor simple.

The distinction of pseudo-complex and strict-complex constraints requires the constraints to be expressed in CNF which can be challenging due to the exponential increase of terms (Kuiter et al., 2022). ■

Fig. 1 shows an excerpt of the *Graph Product Line* (GPL), introduced by Lopez-Herrejon and Batory (Lopez-Herrejon and Batory, 2001) as a standard problem for evaluating SPL methodologies. The feature tree includes 12 features, of which four are abstract and eight are concrete. GPL is the root feature. GraphType, and Weight are mandatory features, thus, are part of all products. Features DFSearch and Alg are optional. If we select the feature Alg, we must at least select one of the algorithms: Connected, MSTKruskal, or Shortest. We may choose either the Directed or Undirected feature regarding the graph type. We must also decide whether the graph is Weighted or UnWeighted. Finally, two complex constraints as propositional formulas express dependencies between features. The first is a pseudo-complex constraint (Shortest ⇒ Directed ∧ Weighted), indicating that selecting the Shortest feature implies selecting both Directed and Weighted features. This constraint can be directly converted into two simple constraints (Shortest ⇒ Directed and Shortest ⇒ Weighted). The second is a strict-complex constraint (Connected ∧ Undirected ⇒ DFSearch), meaning that the feature DFSearch is required when both Connected and Undirected features are selected together. This constraint cannot be directly translated to simple constraints.

The semantics of the feature model FM is its set of products (Schobbens et al., 2007), which consists of the products which satisfy the relations of the feature tree $\mathcal{T}$ and the constraints Ψ. Since we differentiate abstract and concrete features, we need to distinguish between configurations and products of the feature model (Hentze et al., 2022).

Definition 4 (*Configuration, Product*)**.** A *configuration* c of a feature model FM is a subset of its features, that is, $c \in \mathcal{P}(\mathcal{F})$,[1] satisfying the relations of the feature tree and the constraints. We define $\mathcal{CF}$ as the set of all configurations of FM. A *product* p of a feature model FM is a filtered configuration with only the features mapped to implementation artifacts, $p \in \mathcal{P}(\mathcal{F}_c)$. We define $\mathcal{PR}$ as the set of all products of FM. ■

Note that $|\mathcal{PR}| \neq |\mathcal{CF}|$ only when there are abstract features that are leaf features (Hentze et al., 2022), that is, there exists an abstract feature $f \in \mathcal{F}_a$ that does not appear as a parent feature in any relation $r \in \mathcal{R}$. An example of a feature model configuration depicted in Fig. 1 is {GPL, GraphType, UnDirected, Weight, UnWeighted, DFSearch}. An example of a product is {UnDirected, UnWeighted, DFSearch}. The GPL excerpt in Fig. 1 has a total of 36 configurations and products, while the complete version of the GPL has 107,094 configurations and products (see Section 7).

2.2. Relaxed feature models

The *relaxed feature models* were introduced by Knüppel et al. (2017) in their proposal to *eliminate complex constraints*. They developed an algorithm (summarized in Fig. 2) to eliminate complex constraints by incorporating a new set of abstract features and augmenting the number of simple constraints. The resulting feature model (a.k.a. *relaxed feature models*) maintains its semantics (i.e., the products) but not the number of configurations since the resulting model allows abstract features to be leaf features and part of simple constraints. Our Definition 1 of FM supports this. However, for our convenience, in this paper, we adapt the definition of relaxed feature models from Knüppel et al. (2017) to differentiate the new set of auxiliary abstract features from the original set of features.

Definition 5 (*Relaxed Feature Model*)**.** A *relaxed feature model* ($\text{FM}_{\text{relaxed}}$) is a feature model FM where we augment the feature tree with a new set of auxiliary abstract features $\mathcal{F}_{aux}$, and the set of constraints only contains simple constraints (Ψ_s). Formally, $\text{FM}_{\text{relaxed}}$ is defined as a tuple $(\mathcal{T}', \Psi_s)$ with
$\mathcal{T}' = (\mathcal{F}, \mathcal{F}_c, \mathcal{F}_a, \mathcal{F}_{aux}, r, \mathcal{R})$, where

- $\mathcal{F}, \mathcal{F}_c, \mathcal{F}_a, r, \mathcal{R}$ follow Definition 2, and
- $\mathcal{F}_{aux} \subseteq \mathcal{F}_a$ is a new set of auxiliary abstract features. ■

Although an $\text{FM}_{\text{relaxed}}$ can be helpful for the combination of tools and algorithms working with different dialects of the feature model (Knüppel et al., 2017), it poses scalability problems for the AAFM since the size of the feature model increases significantly in features and simple constraints (see Section 7). For instance, the complete GPL FM has 66 features and 32 constraints (28 complex and 4 simple), while the $\text{FM}_{\text{relaxed}}$ version of GPL has 156 features (an increase of 136%) and 86 simple constraints (an increase of 169%).

2.3. Generalized feature trees

The *generalized feature trees* were introduced by van den Broek and Galvão (2009), van den Broek et al. (2008) in their proposal to *eliminate simple constraints*. They propose two functions to help us remove simple constraints, delivering a feature tree where features may have multiple occurrences, a.k.a. *generalized feature tree*. The two functions are the followings:
CommitmentFeature: $\mathcal{T} \times \mathcal{F} \rightarrow \mathcal{T}$ (Algorithm 1). Given a feature tree $\mathcal{T}$ and a feature $f \in \mathcal{F}$ returns the tree $\mathcal{T}(+f)$ whose products contain f.
DeletionFeature: $\mathcal{T} \times \mathcal{F} \rightarrow \mathcal{T}$ (Algorithm 2). Given a feature tree $\mathcal{T}$ and a feature $f \in \mathcal{F}$ returns the feature tree $\mathcal{T}(-f)$ whose products lack f.

[1] $\mathcal{P}(\mathcal{F})$ is the powerset of $\mathcal{F}$ (i.e., the set of all subsets of $\mathcal{F}$).

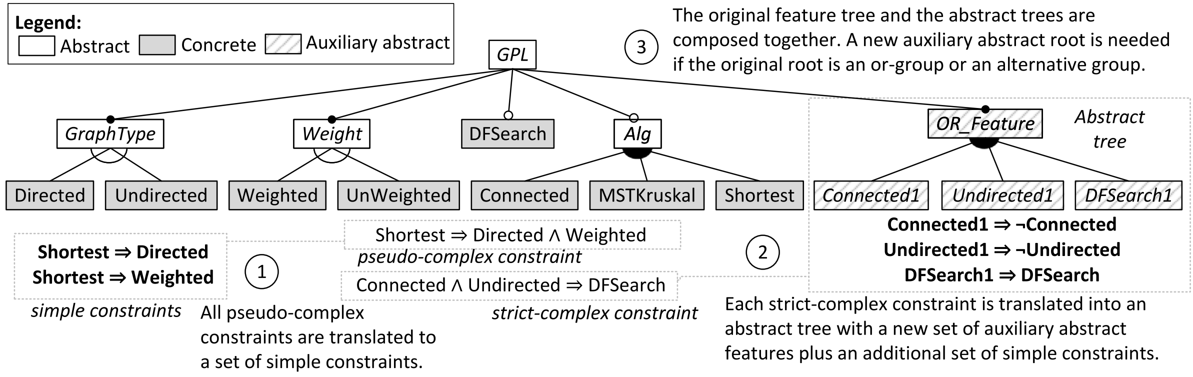


Fig. 2. Algorithm to eliminate complex constraints (Knüppel et al., 2017).

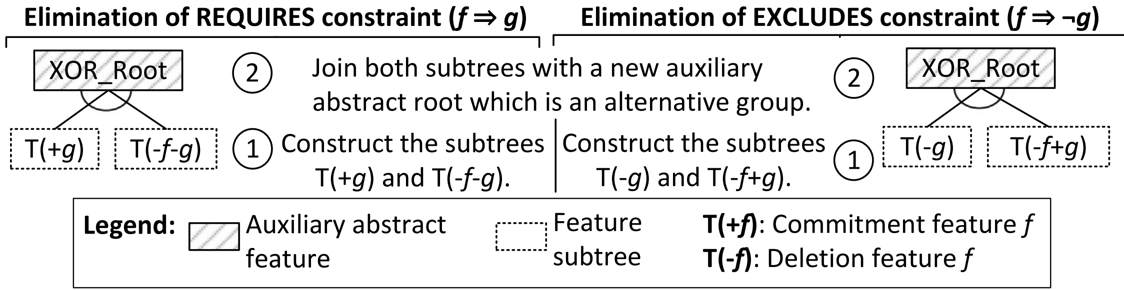


Fig. 3. Algorithms to eliminate simple constraints (van den Broek et al., 2008).

The resulting trees from the application of these functions can be seen as **subtrees** of $\mathcal{T}$ because $\mathcal{T}(+f) \subseteq \mathcal{T}$ and $\mathcal{T}(-f) \subseteq \mathcal{T}$. These functions are applied during the elimination of the *requires* and *excludes* constraints following the steps summarized in Fig. 3. van den Broek and Galvão (2009), van den Broek et al. (2008) proposed applying the algorithms in sequence to eliminate all constraints, considering multiple occurrences of features in the tree. To support this, we provide the following definition.

Definition 6 (*Generalized Feature Tree*). A *generalized feature tree* (GFT) is a feature model FM where the feature tree $\mathcal{T}$ may have multiple occurrences of the same features, and has no constraints (i.e., $\Psi = \emptyset$). Formally, GFT is defined as a 6-tuple $(\mathcal{F}', \mathcal{F}'_c, \mathcal{F}'_a, \mathcal{F}'_{aux}, r, \mathcal{R})$, where

- $\mathcal{F}', \mathcal{F}'_c, \mathcal{F}'_a, \mathcal{F}'_{aux}$ follow Definition 5, but using lists instead of sets, so that features may have multiple occurrences. They are only distinguishable from each other due to their relations $\mathcal{R}$.
- $r, \mathcal{R}$ follow Definition 2. ■

Note that $\mathcal{F}'_{aux}$ now also contains the new auxiliary abstract roots (xor-groups) used to eliminate each constraint (see Fig. 3). Note also that in a GFT, multiple occurrences are in different subtrees of an alternative (xor) group, so products do not have multiple occurrences of features. The problem with Broek's algorithms is that eliminating a constraint in the worst case doubles the size of the feature tree. Therefore, the size of the resulting GFT is exponential in the number of constraints in the worst case. Moreover, van den Broek and Galvão (2009), van den Broek et al. (2008) do not consider complex constraints. Thus, the result is even worse when applying the algorithms to an $\text{FM}_{\text{relaxed}}$. For instance, while the $\text{FM}_{\text{relaxed}}$ of GPL has 156 features and 86 simple constraints, the GFT of GPL has 2,733,485 features.

In the following, we present our approach that uses the concepts introduced in this section to eliminate all constraints (complex and simple) and analyze a GFT without generating it completely.

3. Idea and approach overview

The idea of our approach relies on the following three theorems:

Algorithm 1 Commitment feature (adapted from van den Broek et al., 2008).

Input: $\mathcal{T}$: The feature tree; f: A feature ($f \in \mathcal{F}$).
Output: $\mathcal{T}(+f)$: The feature tree whose products are those products of $\mathcal{T}$ which contain f.

1: **function** CommitmentFeature($\mathcal{T}$, f)
2: **if** $\mathcal{T} = NIL \lor f \notin \mathcal{T}.\mathcal{F}$ **then return** NIL ▷ $\mathcal{T}$ is empty or does not contain f.
3: **else**
4: **while** $f \neq \mathcal{T}.r$ **do** ▷ f is not the root.
5: $p \leftarrow$ `parent`(f)
6: **if** $\neg$ `is_group`(p) $\land$ `is_optional`(f) **then**
7: `relation`(p,f)`.card_min` $\leftarrow 1$ ▷ Make f a mandatory subfeature of p.
8: **else if** `is_xor_group`(p) **then** ▷ Make f a single mandatory subfeature of p and remove other subfeatures of p.
9: $subfeatures \leftarrow$ `relation`(p,f)`.children` $\setminus \{f\}$
10: `relation`(p,f)`.children` $\leftarrow \{f\}$
11: **for all** $c \in subfeatures$ **do**
12: $\mathcal{T} \leftarrow$ DeleteFeatureBranch($\mathcal{T}$, c)
13: **end for**
14: **else if** `is_or_group`(p) **then** ▷ Make f mandatory and other subfeatures of p optional.
15: $children \leftarrow$ `relation`(p,f)`.children`
16: $\mathcal{T}.\mathcal{R} \leftarrow \mathcal{T}.\mathcal{R} \setminus \{$`relation`$(p,f)\}$
17: $\mathcal{T}.\mathcal{R} \leftarrow \mathcal{T}.\mathcal{R} \bigcup \{(p,[f],\langle 1..1\rangle)\}$
18: $\mathcal{T}.\mathcal{R} \leftarrow \mathcal{T}.\mathcal{R} \bigcup \{(p,[g],\langle 0..1\rangle)\}, \forall g \in children | g \neq f\}$
19: **end if**
20: $f \leftarrow p$ ▷ Repeat with p instead of f.
21: **end while**
22: **end if**
23: **return** $\mathcal{T}$
24: **end function**
25: **function** DeleteFeatureBranch($\mathcal{T}$, f)
26: $children \leftarrow \{g \in r.$`children`$, \forall r \in \mathcal{T}.\mathcal{R} | r.$`parent`$= f\}$
27: $relations \leftarrow \{r \in \mathcal{T}.\mathcal{R} | f = r.$`parent`$\}$
28: $\mathcal{T}.\mathcal{F} \leftarrow \mathcal{T}.\mathcal{F} \setminus \{f\}$
29: $\mathcal{T}.\mathcal{R} \leftarrow \mathcal{T}.\mathcal{R} \setminus relations$
30: **for all** $c \in children$ **do**
31: $\mathcal{T} \leftarrow$ DeleteFeatureBranch($\mathcal{T}$, c)
32: **end for**
33: **return** $\mathcal{T}$
34: **end function**

Theorem 1 (*Disjoint Semantics for Subtrees*). *The elimination of a simple constraint from a feature model FM results in two subtrees of $\mathcal{T}$ with disjoint*

Algorithm 2 Deletion feature (adapted from van den Broek et al., 2008).

```
Input: T: The feature tree; f: A feature (f ∈ F).
Output: T(−f): The feature tree whose products are those products of T without f.
 1: function DeletionFeature(T, f)
 2:   if T = NIL then return NIL                          ▷ Returns the empty tree.
 3:   else if f ∉ T.F then return T                       ▷ T does not contain f.
 4:   else
 5:     p ← parent(f)
 6:     while f ≠ T.r ∧ ¬is_group(p) ∧ is_mandatory(f) do
 7:       f ← p
 8:       p ← parent(f)
 9:     end while
10:     if f = T.r then return NIL                        ▷ Returns the empty tree.
11:     else if ¬ is_group(p) ∧ is_optional(f) then        ▷ f is an optional
        subfeatures of p.
12:       T.R ← T.R \ {relation(p,f)}
13:       T ← DeleteFeatureBranch(T, f)                   ▷ Delete feature f.
14:     else if is_group(p) then
         ▷ Delete f from the group; if p has only one remaining subfeature, make it mandatory.
15:       if relation(p,f).card_max > 1 then
16:         relation(p,f).card_max ← relation(p,f).card_max −1
17:       end if
18:       relation(p,f).children ← {f}
19:       T ← DeleteFeatureBranch(T, f)                   ▷ Delete feature f.
20:     end if
21:   end if
22:   return T
23: end function
```

semantics (i.e., the products of $\mathcal{T}$ are divided into two disjoint sets of products).

Proof. Eliminating a simple constraint implies the application of two alternative transformations to the feature tree $\mathcal{T}$: $Rt_0 = \mathcal{T}(+g)$ and $Rt_1 = \mathcal{T}(-f-g)$ for requires constraints ($f \Rightarrow g$), or $Et_0 = \mathcal{T}(-g)$ and $Et_1 = \mathcal{T}(-f+g)$ for excludes constraints ($f \Rightarrow \neg g$) (based on the CommitmentFeature and DeletionFeature functions presented in Section 2). Independently from the constraint type (requires or excludes), the transformations t_0 and t_1 are mutually exclusive because one transformation always forces the feature g to be present in all products ($\mathcal{T}(+g)$) while the other transformation removes the same feature g from all products ($\mathcal{T}(-g)$). Thus, the two resulting subtrees of $\mathcal{T}$ contain a different disjoint set of products. ■

Theorem 2 (*Smaller Size for Subtrees*). *The elimination of a simple constraint from a feature model FM results in two subtrees of $\mathcal{T}$. The size of those subtrees (i.e., the number of features) is always less than or equal to the size of $\mathcal{T}$.*

Proof. Eliminating a requires constraint results in the following two subtrees of $\mathcal{T}$: $R_0 = \mathcal{T}(+g)$ and $R_1 = \mathcal{T}(-f-g)$. To obtain R_0 the CommitmentFeature function needs to be applied over feature g which implies converting the feature g, and its parents recursively up to the root, into mandatory features (see Algorithm 1). The CommitmentFeature function only modifies the size of the tree if any of the parent-features is an alternative (xor) group, in which case the branches of the siblings of the converted mandatory feature will be deleted. To obtain R_1, the DeletionFeature function needs to be applied over feature g which implies removing the feature g from the tree along with all its children and mandatory parents recursively. Similarly, eliminating an excludes constraint results in the following two subtrees: $E_0 = \mathcal{T}(-g)$ and $E_1 = \mathcal{T}(-f+g)$. In both cases, to obtain E_0 and E_1, the DeletionFeature function needs to be applied, removing the appropriate features, and thus reducing the size of the resulting subtree. ■

Theorem 3 (*Empty Subtrees*). *The application of the CommitmentFeature and the DeletionFeature functions over the same feature in a sequence results in an empty tree. That is: $\mathcal{T}(+f-f) = \mathcal{T}(-f+f) = NIL, \forall f \in \mathcal{T}$.*

Proof. We first demonstrate that $\mathcal{T}(+f-f) = NIL$. Applying the CommitmentFeature function to f means making feature f mandatory as well as making mandatory all its parents recursively up to the root feature (application of Algorithm 1). Then eliminating such a feature f means eliminating all features from f to the root feature (see lines 6 to 10 in Algorithm 2) resulting in an empty tree (line 10 in Algorithm 2). To demonstrate $\mathcal{T}(-f+f) = NIL$, we first apply Algorithm 2 over feature f which results in a feature tree without feature f. Then, since feature f is not anymore in the tree, applying the CommitmentFeature function to f results in an empty tree (see line 2 in Algorithm 1). ■

Theorem 1 allows us to manage the resulting subtrees independently. When eliminating a simple constraint, instead of joining the subtrees with an additional xor-group root feature (as shown in Fig. 3), we can maintain both subtrees separate as they were different feature models and remove the subsequent constraints from both subtrees independently. Repeating the process for every simple constraint will result in 2^n different subtrees, n being the number of simple constraints in the feature model. Theorems 2 and 3 allow us to deal with the complexity in space given by the large number of possible subtrees (2^n). The application of the CommitmentFeature and DeletionFeature functions normally reduces the size of the feature tree because one or more features are often removed when committing or deleting a feature (Theorem 2). Moreover, when applying those functions in sequence to eliminate all constraints, most of the resulting subtrees will be empty trees, especially when the constraints share the same features (Theorem 3). The other non-empty subtrees are considered valid. A subtree is valid if it represents at least one product (i.e., it is a non-empty feature tree).

Fig. 4 provides an overview of our approach for eliminating all constraints from a feature model (FM).

The process is divided into two main parts plus an optional third part: (i) a first part in which we obtain an intermediate representation of the feature model (called $\mathrm{FM_{Sans}}$) that can be seen as a new *knowledge compilation technique* (Darwiche and Marquis, 2002; Mendonça et al., 2008) of the original feature model (steps 1 to 3 in Fig. 4); (ii) a second part in which we process the $\mathrm{FM_{Sans}}$ to analyze it exploiting parallelization techniques (steps 4 to 6); and (iii) an additional third optional part in which we obtain a generalized feature tree (GFT) and analyze it (steps 7 and 8). The steps of our approach are summarized as follows. Notice that this is an overview of our approach, and thus, each step will be further explained in Sections 4 to 6. The starting point is a feature model FM that conforms to Definition 1.

(I) knowledge compilation technique for constraints removal: FM_{Sans} construction (top of Fig. 4). Steps 1 to 3 are in charge of constructing a new representation of the feature model where constraints are codified to be efficiently removed.

Step 1. Eliminate complex constraints. The first step consists of eliminating complex constraints by applying the algorithm proposed by Knüppel et al. (2017) to obtain a relaxed feature model ($\mathrm{FM_{relaxed}}$) that only contains simple constraints. The formalization and details of such an algorithm are beyond the scope of this paper (Knüppel et al., 2017; Knüppel, 2016), but for completeness and self-containing, the algorithm is summarized in Fig. 2 using our running example. As shown in Fig. 2, after eliminating the two complex constraints existing in the GPL example, the resulting $\mathrm{FM_{relaxed}}$ now comprises five simple constraints and four new auxiliary abstract features. We have improved Knüppel's algorithm to avoid an additional auxiliary synthetic root feature, used to compose the original feature tree and the abstract trees, which is only needed when the original root feature is a group (see step 3 in Fig. 2).

Step 2. Codification of simple constraints for their removal. The input of this step is a $\mathrm{FM_{relaxed}}$, that is, a feature model with only simple constraints (requires and excludes).

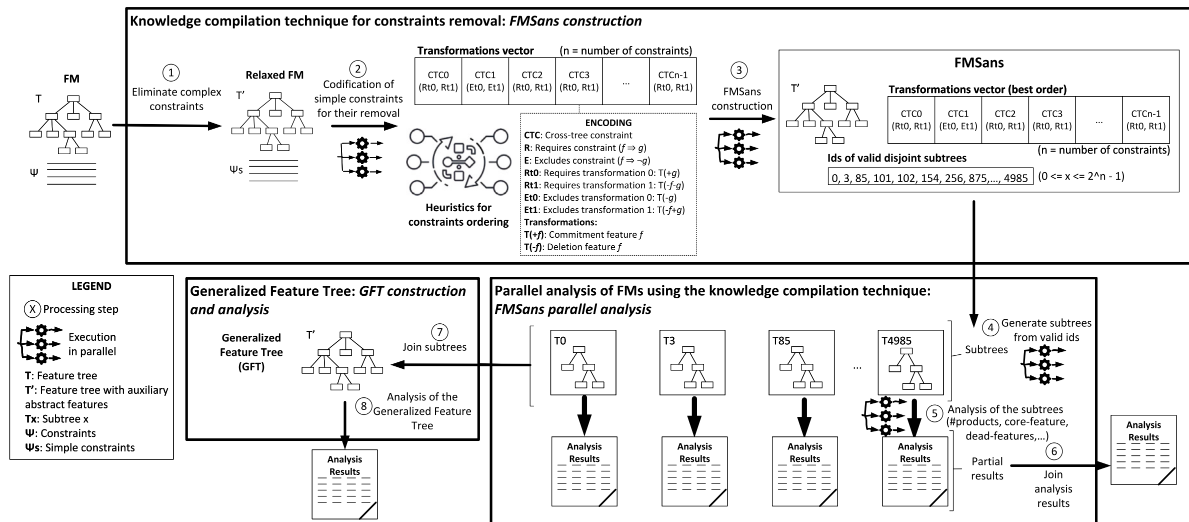


Fig. 4. Our approach for the complete elimination of cross-tree constraints and the parallel analysis of feature models.

The goal of this step is to obtain a codification of the transformations that allows removing the simple constraints from the feature model by avoiding the space complexity of the existing approach (the Broek's algorithm) (van den Broek et al., 2008). Taking into account Theorem 1 (Disjoint semantics for subtrees), we define a `Transformations vector` that encodes the sequences of transformations needed to eliminate all the constraints. An instance of the transformations vector is a sequence of 0's and 1's indicating which transformation will be executed for each constraint. The order in which the constraints are codified in the transformations vector may affect the number of instances of the vector that need to be executed to obtain all valid subtrees. Therefore, in this step, we also analyze different heuristics for ordering the constraints in the transformations vector to reduce the number of transformations to be executed. The details of this step are given in Section 4.

Step 3. FM_{Sans} **construction.** This step takes a transformations vector, that codifies the transformations needed to eliminate all simple constraints, and builds an FM_{Sans}. Given a transformations vector, we need to identify the instances of the vector in which its execution results in valid subtrees. The subtrees themselves are not stored in memory because of their complexity in size (i.e., the number of valid subtrees can be huge). Thus, our FM_{Sans} only stores the identifiers (a natural number) of those instances of the transformations vector that reach a valid subtree (e.g., the valid subtree with $id = 3$ codifies the instance 00011 of the transformations vector). As in Broek's approach (van den Broek and Galvão, 2009; van den Broek et al., 2008), even though the complexity of executing the transformations vector is linear, the exponential computational complexity in time to identify all valid subtrees is inevitable. However, The codification performed in the previous step 2 enables our approach to be fully parallelizable since each instance of the transformations vector is entirely independent of the other (Theorem 1). The details of this step are given in Section 5.

(II) parallel analysis of FMs using the knowledge compilation technique: FM_{Sans} parallel analysis (bottom-right of Fig. 4). Steps 4 to 5 focus on the parallel analysis of the FM_{Sans}.

Step 4. Generate subtrees from valid ids. As previously described, the FM_{Sans} model stores the identifiers of the instances of the transformations vector that reach a valid subtree. Executing those instances we directly obtain all subtrees of $\mathcal{T}$ without constraints. Subtrees are entirely independent of each other, and they can be generated and analyzed by exploiting parallel techniques. The details of this and the following steps are given in Section 6.

Step 5. Analysis of the subtrees in parallel. The subtrees can be analyzed individually and thus in parallel, as they are independent non-related feature models without constraints. Here the benefit of our approach is threefold: (1) each feature subtree is a subtree of the original feature tree $\mathcal{T}'$ and thus smaller in size (containing fewer features) by Theorem 2; (2) there are no constraints which thus facilitates the analysis; and (3) each subtree is completely independent of the others and thus can be analyzed in parallel by Theorem 1. In our approach, we can perform several analysis operations at once (e.g., valid, products, number of products, core features, dead features,…); after the analysis, the subtrees are discarded to free up memory.

Step 6. Join analysis results. The analysis results of each subtree are independent of each other and must be joined to obtain the final result of the complete feature model. In this step, we define join operations for the main analysis operations on feature models (Benavides et al., 2010).

(III) generalized feature tree: $\mathcal{GFT}$ construction and analysis (bottom-left of Fig. 4). Steps 7 and 8 are optional and allow us to generate and analyze the complete GFT by joining all subtrees.

Step 7. Join subtrees to generate a $\mathcal{GFT}$. An alternative step of our approach when building the subtrees in step 4 is to maintain and join all subtrees to generate the complete GFT, as proposed by van den Broek and Galvão (2009), van den Broek et al. (2008). However, the GFT presents several issues. First, it may be impossible to build due to its size in the number of features (see Section 7). Second, the resulting GFT, in general, cannot be directly analyzed with SAT solvers because of multiple occurrences of features, even if the duplicated features occur as subfeatures of an alternative (xor) group. Our contribution here is that we join all subtrees with only one auxiliary abstract feature as the root feature, an alternative group, in contrast to incorporating one auxiliary abstract feature for each constraint removal as in van den Broek et al. (2008).

Step 8. Analysis of the Generalized Feature Tree. The GFT can be directly analyzed using the hierarchy structure of the feature tree (van den Broek and Galvão, 2009), which has the advantage that for small feature models, the analysis operations outperform the analysis with BDD solvers (van den Broek and Galvão, 2009) (see Section 7).

In the following sections, we focus on the codification of simple constraints and the ordering of the transformations vector to obtain the FM_{Sans} (step 2 - Section 4), on the generation of the valid subtrees using parallelism (step 3 - Section 5), and on the analysis of the FM_{Sans} (steps 4 et seq.- Section 6). A comparison of our approach with existing

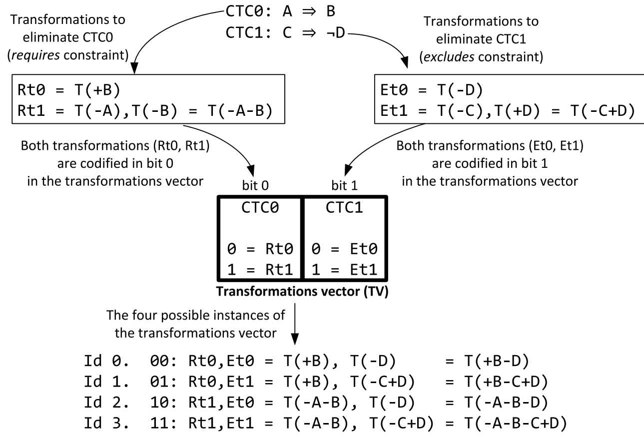


Fig. 5. Codification of two simple constraints in the transformations vector.

approaches for the elimination of constraints in feature models can be found in Table 7 in Section 8.

4. Codification of simple constraints for their removal

This section provides the details of step 2 in our approach, which takes a feature model with only simple constraints (an $FM_{relaxed}$) as input and generates an encoding for efficient removal of the constraints.

4.1. Binary encoding for constraints removal

We define the transformations vector that codifies the transformations needed to eliminate all constraints from the feature model as follows:

Definition 7 (*Transformations Vector $\mathcal{TV}$*). A transformations vector $\mathcal{TV}$ is a sequence of encoding transformations of length n, where n is the number of constraints. Each component represents the encoding of a simple constraint $CTC_i \in \mathcal{TV}, 0 \leq i < n$ including the two alternative transformations to eliminate it: Rt_0 and Rt_1 for *requires* constraints ($f \rightarrow g | f, g \in \mathcal{F}$), or Et_0 and Et_1 for *excludes* constraints ($f \rightarrow \neg g | f, g \in \mathcal{F}$):

- $\boldsymbol{Rt_0}$ is the transformation in charge of constructing $\mathcal{T}(+g)$, which consists in applying the `CommitmentFeature` operation with feature g.
- $\boldsymbol{Rt_1}$ is the transformation in charge of constructing $\mathcal{T}(-f-g)$, which consists in applying the `DeletionFeature` operation with features f and g consecutively.
- $\boldsymbol{Et_0}$ is the transformation in charge of constructing $\mathcal{T}(-g)$, which consists in applying the `DeletionFeature` operation with feature g.
- $\boldsymbol{Et_1}$ is the transformation in charge of constructing $\mathcal{T}(-f+g)$, which consists in applying the `DeletionFeature` operation with feature f and then the `CommitmentFeature` operation with feature g.

An instance $v_x \in \mathcal{TV}$ is a vector of 0's and 1's indicating which transformation (t_0 or t_1) will be executed for each constraint. The suffix (0 or 1) indicates the transformation (t_0 or t_1) to be executed according to the value of bit i in the instance $v_x \in \mathcal{TV}$. ■

Before going into more details, let us illustrate the concept of the transformations vector $\mathcal{TV}$ with an easy example to understand the codification proposed (see Fig. 5). Given two simple constraints (CTC0 and CTC1) where CTC0 is a *requires* constraint, and CTC1 is a *excludes* constraint. The transformations vector ($\mathcal{TV}$) has length 2 because there are 2 constraints. The first bit (bit 0) of $\mathcal{TV}$ codifies the two transformations (Rt0 and Rt1) in charge of eliminating the first constraint (CTC0), while the second bit (bit 1) of $\mathcal{TV}$ codifies the two transformations (Et0 and Et1) in charge of eliminating the second constraint (CTC1). The value of each bit indicates which specific transformation will be executed. Therefore, there are four possible instances of the transformations vector in Fig. 5: 00, 01, 10, and 11. For example, the instance 10 indicates that the transformation Rt1 (bit 0 is 1) following with the transformation Et0 (bit 1 is 0) will be executed.

Fig. 6 shows the transformations vector $\mathcal{TV}$ for the GPL running example encoding the five simple constraints. There are three *requires* constraints (CTC_0, CTC_1, CTC_4), and two *excludes* constraints (CTC_2, CTC_3). For example, CTC_0 codifies the *requires* constraint `Shortest` ⇒ `Directed` with the two alternative transformations $Rt_0 = \mathcal{T}(+\texttt{Directed})$ and $Rt_1 = \mathcal{T}(-\texttt{Shortest}-\texttt{Directed})$.

The execution of a binary vector $v_x \in \mathcal{TV}$ involves applying all transformations codified in the transformations vector $\mathcal{TV}$ to the original feature tree $\mathcal{T}$ according to the binary value of x (see right-hand side of Fig. 6). The transformations are applied in sequence, starting from the transformation codified in bit 0 and ending at the transformation codified in the last bit ($n-1$). The execution of a vector v_x finishes as soon as a transformation results in an empty tree or when all transformations are executed resulting in a valid subtree. When the value of the bit $i \in v_x$ is 0, we execute the transformation Rt_0 or Et_0, depending on whether the constraint codified in the bit i is a *requires* or an *excludes* constraint, respectively. When the value of the bit $i \in v_x$ is 1, we execute the transformation Rt_1 or Et_1. For example, a possible instance of $\mathcal{TV}$ is 10010 ($id = 18$) indicating that we need to execute the transformation Rt_1 first over the feature tree $\mathcal{T}$ of the feature model, then Et_0 over the resulting subtree, and so on; resulting in a valid subtree (subtree with the identifier 18 in the bottom-right of Fig. 6). The complexity of executing a vector v_x is linear in the number of constraints (van den Broek et al., 2008).

4.2. Constraints removal heuristics for identifying valid subtrees

The codification given by the transformations vectors $\mathcal{TV}$ leads to 2^n instances $v_x \in \mathcal{TV}$. However, according to Theorems 2 and 3, not all instances result in valid subtrees (see red crosses at the right-hand side of Fig. 6).

During the execution of an instance v_x of the transformations vector $\mathcal{TV}$, when the transformation associated with the bit i results in an empty subtree (NIL), the remaining transformations associated with the bits to the right in the $\mathcal{TV}$ do not need to be executed regardless of their values (because all resulting subtrees will be empty). Therefore, we can skip the execution of $2^{n-i}-1$ vectors. The earlier the empty subtree is found (i.e., the further to the left in $\mathcal{TV}$), the higher the number of transformations that reach invalid (empty) subtrees are skipped for being executed, reducing the number of instances of $\mathcal{TV}$ to be executed. In our example (right-hand side of Fig. 6), the binary vector $Id2$ results in an empty subtree after the execution of bit 3; thus, the transformation associated with bit 4 does not need to be executed regardless of its value and the following binary vector ($Id3$) is skipped.

The challenge at this point is to obtain an ordering for the constraints in the transformations vector $\mathcal{TV}$ so that the number of instances of the transformations vector that need to be executed to identify the valid subtrees is minimal. We are following the same approach used in the construction of BDDs, where the size of BDDs is very susceptible to the input variable order, and heuristics have been defined to cope with this scalability problem (Thüm, 2020; Popov et al., 2023).

We propose four different heuristics classified into two types:

- Heuristics that use the knowledge provided by the specific instance of the feature model being analyzed.
- Heuristics that are independent of the specific instance of the feature model.

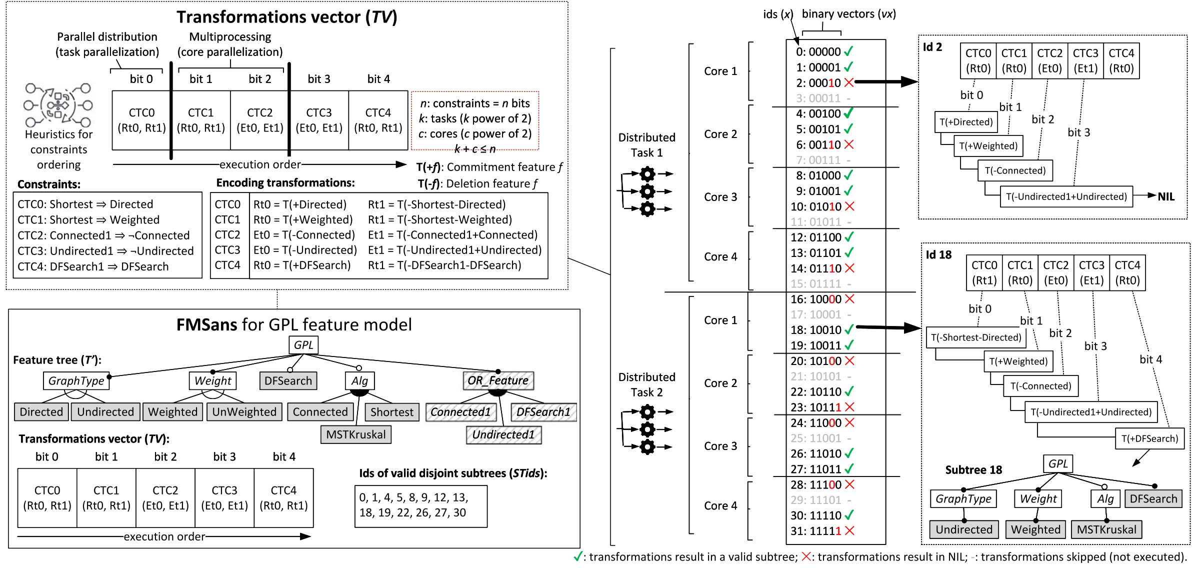


Fig. 6. Transformations vector $\mathcal{TV}$ for the GPL (top-left) encoding the transformations to eliminate the constraints. The parallel process (right) to obtain the valid instances of the transformations vector and the associated valid subtrees. The resulting FM_{Sans} model (bottom-left) with the feature tree and the set of identifiers of the valid subtrees of the transformations vector.

4.2.1. Heuristics with knowledge of the feature model instance

These heuristics use knowledge about the specific feature model instance being analyzed.

H1. Normal order heuristic (NOH). It maintains the order given by the original feature model. This order may be defined by the domain experts who designed the original feature model, which is the case of small and medium-size feature models. It may also be automatically generated by the program in charge of generating the feature model, which is the case for large-scale feature models (e.g., Automotive, Linux feature models). In any case, note that the order of simple constraints may differ from the original order because the heuristics consider the simple constraints after removing complex constraints, which add new simple constraints (see Step 1 in our approach). The normal order is useful as a baseline for comparison with other heuristics (see Section 7).

H2. Feature tree size heuristic (FTSH). This heuristic is based on Theorem 2, which considers the smaller size of the resulting feature tree after applying the transformation responsible for eliminating a constraint. It sorts the constraints by putting those for which the transformations that remove them result in a smaller subtree first. These are the transformations that remove more features. Given a constraint $c \in \Psi_s$, this heuristic calculates the number of features that will be removed by each of the alternative transformations applied over the feature tree $\mathcal{T}$ according to Eq. (1) for transformation t_0 and Eq. (2) for transformation t_1. The equations rely on the functions $FTSH_{Commit}(\mathcal{T}, f)$ and $FTSH_{Delete}(\mathcal{T}, f)$ that calculate the number of features that are removed from $\mathcal{T}$ when applying the `CommitmentFeature` and the `DeletionFeature` functions, respectively. The auxiliary equation $\mathcal{ST}_{size}(\mathcal{T}, f)$ calculates the size of the subtree of feature f as root.

$$FTSH_{t_0}(\mathcal{T}, c) = \begin{cases} FTSH_{Commit}(\mathcal{T}, g), & \text{if } c \in \Psi_{s_R} \mid c = f \rightarrow g \\ FTSH_{Delete}(\mathcal{T}, g), & \text{if } c \in \Psi_{s_E} \mid c = f \rightarrow \neg g \end{cases} \tag{1}$$

$$FTSH_{t_1}(\mathcal{T}, c) = \begin{cases} FTSH_{Delete}(\mathcal{T}, f) + FTSH_{Delete}(\mathcal{T}, g), & \text{if } c \in \Psi_{s_R} \mid c = f \rightarrow g \\ FTSH_{Delete}(\mathcal{T}, f) + FTSH_{Commit}(\mathcal{T}, g), & \text{if } c \in \Psi_{s_E} \mid c = f \rightarrow \neg g \end{cases} \tag{2}$$

$$FTSH_{Commit}(\mathcal{T}, f) = \begin{cases} 0, & \text{if } f = \mathcal{T}.r \\ FTSH_{Commit}(\mathcal{T}, parent(f)) + \mathcal{ST}_{size}(\mathcal{T}, g) \\ \quad \forall g \in relation(f, g).children \\ \quad\quad \mid f \neq g, & \text{if } is_xor_group(parent(f)) \\ FTSH_{Commit}(\mathcal{T}, parent(f)), & \text{otherwise} \end{cases}$$

$$FTSH_{Delete}(\mathcal{T}, f) = \begin{cases} |\mathcal{T}|, & \text{if } f = \mathcal{T}.r \\ FTSH_{Delete}(\mathcal{T}, parent(f)), & \text{if } is_mandatory(f) \\ \mathcal{ST}_{size}(\mathcal{T}, f), & \text{otherwise} \end{cases}$$

$$\mathcal{ST}_{size}(\mathcal{T}, f) = \begin{cases} 1, & \text{if } is_leaf(f) \\ 1 + \mathcal{ST}_{size}(\mathcal{T}, g) & \forall g \in relation(f, g).children, \forall r \in \mathcal{T}.\mathcal{R} \\ 0, & \text{otherwise} \end{cases}$$

The transformations vector is sorted by following a greedy approach, in which the heuristic is applied to each constraint individually considering the original complete feature tree $\mathcal{T}$. Moreover, for each constraint, it also considers swapping the order of the transformation t_0 and t_1 by putting first in t_0 the transformation that removes more features because transformation t_0 will be executed first in the vector. In this way, a bit with value 0 in an instance vector $v_x \in \mathcal{TV}$ can execute the transformation t_0 or t_1, and a bit with value 1 may execute the transformation t_0 or t_1 based on the transformation that removes more features.

4.2.2. Heuristics independent of the feature model knowledge

These heuristics are generic and do not use information contained within the feature model instances.

H3. Random heuristic (RH). This heuristic sorts the constraints of the transformations vector $\mathcal{TV}$ randomly. Moreover, the random heuristic also shuffles the two alternative transformations for each constraint (i.e., Rt_0 and Rt_1 for *requires* constraints and Et_0 and Et_1 for *excludes* constraints) in the binary codification t_0 and t_1. This way, a bit with a value 0 in an instance vector $v_x \in \mathcal{TV}$ may execute the transformation t_1, and a bit with value 1 may execute the transformation t_0.

H4. Feature shared in constraints heuristic (FSCH). This heuristic considers the relationships between the constraints to be eliminated by taking into account the features that are shared across multiple constraints. To better illustrate this heuristic, Table 1 shows the constraints' analysis

Table 1
Constraints' analysis for features sharing with two simple constraints.

Rule	Case	Type	Order 1	Transformations	Valid*	Order 2	Transformations	Valid*	Conclusions analysis
Feature shared in the antecedent	C1	REQUIRES	CTC_0: $B \to A$	$Rt_0Rt_0 = \mathcal{T}(+A+C)$	✓	CTC_0: $B \to C$	$Rt_0Rt_0 = \mathcal{T}(+C+A)$	✓	Indifferent order. There are no empty trees.
			CTC_1: $B \to C$	$Rt_0Rt_1 = \mathcal{T}(+A-B-C)$	✓	CTC_1: $B \to A$	$Rt_0Rt_1 = \mathcal{T}(+C-B-A)$	✓	
				$Rt_1Rt_0 = \mathcal{T}(-B-A+C)$	✓		$Rt_1Rt_0 = \mathcal{T}(-B-C+A)$	✓	
				$Rt_1Rt_1 = \mathcal{T}(-B-A-B-C)$	✓		$Rt_1Rt_1 = \mathcal{T}(-B-C-B-A)$	✓	
	C2	EXCLUDES	CTC_0: $B \to \neg A$	$Et_0Et_0 = \mathcal{T}(-A-C)$	✓	CTC_0: $B \to \neg C$	$Et_0Et_0 = \mathcal{T}(-C-A)$	✓	Indifferent order. There are no empty trees.
			CTC_1: $B \to \neg C$	$Et_0Et_1 = \mathcal{T}(-A-B+C)$	✓	CTC_1: $B \to \neg A$	$Et_0Et_1 = \mathcal{T}(-C-B+A)$	✓	
				$Et_1Et_0 = \mathcal{T}(-B+A-C)$	✓		$Et_1Et_0 = \mathcal{T}(-B+C-A)$	✓	
				$Et_1Et_1 = \mathcal{T}(-B+A-B+C)$	✓		$Et_1Et_1 = \mathcal{T}(-B+C-B+A)$	✓	
	C3	REQUIRES and EXCLUDES	CTC_0: $B \to A$	$Rt_0Et_0 = \mathcal{T}(+A-C)$	✓	CTC_0: $B \to \neg C$	$Et_0Rt_0 = \mathcal{T}(-C+A)$	✓	Indifferent order. There are no empty trees.
			CTC_1: $B \to \neg C$	$Rt_0Et_1 = \mathcal{T}(+A-B+C)$	✓	CTC_1: $B \to A$	$Et_0Rt_1 = \mathcal{T}(-C-B-A)$	✓	
				$Rt_1Et_0 = \mathcal{T}(-B-A-C)$	✓		$Et_1Rt_0 = \mathcal{T}(-B+C+A)$	✓	
				$Rt_1Et_1 = \mathcal{T}(-B-A-B+C)$	✓		$Et_1Rt_1 = \mathcal{T}(-B+C-B-A)$	✓	
Feature shared in the consequent	C4	REQUIRES	CTC_0: $A \to B$	$Rt_0Rt_0 = \mathcal{T}(+B+B)$	✓	CTC_0: $C \to B$	$Rt_0Rt_0 = \mathcal{T}(+B+B)$	✓	Indifferent order. Constraints should be grouped by shared features.
			CTC_1: $C \to B$	$Rt_0Rt_1 = \mathcal{T}(+B-A-B)$	✗	CTC_1: $A \to B$	$Rt_0Rt_1 = \mathcal{T}(+B-A-B)$	✗	
				$Rt_1Rt_0 = \mathcal{T}(-A-B+B)$	✗		$Rt_1Rt_0 = \mathcal{T}(-C-B+B)$	✗	
				$Rt_1Rt_1 = \mathcal{T}(-A-B-A-B)$	✓		$Rt_1Rt_1 = \mathcal{T}(-C-B-A-B)$	✓	
	C5	EXCLUDES	CTC_0: $A \to \neg B$	$Et_0Et_0 = \mathcal{T}(-B-B)$	✓	CTC_0: $C \to \neg B$	$Et_0Et_0 = \mathcal{T}(-B-B)$	✓	Indifferent order. Constraints should be grouped by shared features.
			CTC_1: $C \to \neg B$	$Et_0Et_1 = \mathcal{T}(-B-C+B)$	✗	CTC_1: $A \to \neg B$	$Et_0Et_1 = \mathcal{T}(-B-A+B)$	✗	
				$Et_1Et_0 = \mathcal{T}(-A+B-B)$	✗		$Et_1Et_0 = \mathcal{T}(-C+B-B)$	✗	
				$Et_1Et_1 = \mathcal{T}(-A+B-C+B)$	✓		$Et_1Et_1 = \mathcal{T}(-C+B-A+B)$	✓	
	C6	REQUIRES and EXCLUDES	CTC_0: $A \to B$	$Rt_0Et_0 = \mathcal{T}(+B-B)$	✗	CTC_0: $C \to \neg B$	$Et_0Rt_0 = \mathcal{T}(-B+B)$	✗	Indifferent order. Constraints should be grouped by shared features.
			CTC_1: $C \to \neg B$	$Rt_0Et_1 = \mathcal{T}(+B-C+B)$	✓	CTC_1: $A \to B$	$Et_0Rt_1 = \mathcal{T}(-B-A-B)$	✓	
				$Rt_1Et_0 = \mathcal{T}(-A-B-B)$	✓		$Et_1Rt_0 = \mathcal{T}(-C+B+B)$	✓	
				$Rt_1Et_1 = \mathcal{T}(-A-B-C+B)$	✗		$Et_1Rt_1 = \mathcal{T}(-C+B-A-B)$	✗	
Feature shared in the antecedent/consequent	C7	REQUIRES	CTC_0: $A \to B$	$Rt_0Rt_0 = \mathcal{T}(+B+C)$	✓	CTC_0: $B \to C$	$Rt_0Rt_0 = \mathcal{T}(+C+B)$	✓	First the constraint with the shared feature in the consequent. Follow order 1.
			CTC_1: $B \to C$	$Rt_0Rt_1 = \mathcal{T}(+B-B-C)$	✗	CTC_1: $A \to B$	$Rt_0Rt_1 = \mathcal{T}(+C-A-B)$	✓	
				$Rt_1Rt_0 = \mathcal{T}(-A-B+C)$	✓		$Rt_1Rt_0 = \mathcal{T}(-B-C+B)$	✗	
				$Rt_1Rt_1 = \mathcal{T}(-A-B-B-C)$	✓		$Rt_1Rt_1 = \mathcal{T}(-B-C-A-B)$	✓	
	C8	EXCLUDES	CTC_0: $A \to \neg B$	$Et_0Et_0 = \mathcal{T}(-B-C)$	✓	CTC_0: $B \to \neg C$	$Et_0Et_0 = \mathcal{T}(-C-B)$	✓	First the constraint with the shared feature in the consequent. Follow order 1.
			CTC_1: $B \to \neg C$	$Et_0Et_1 = \mathcal{T}(-B-B+C)$	✓	CTC_1: $A \to \neg B$	$Et_0Et_1 = \mathcal{T}(-C-A+B)$	✓	
				$Et_1Et_0 = \mathcal{T}(-A+B-B)$	✓		$Et_1Et_0 = \mathcal{T}(-B+C-B)$	✓	
				$Et_1Et_1 = \mathcal{T}(-A+B-B+C)$	✗		$Et_1Et_1 = \mathcal{T}(-B+C-A+B)$	✗	
	C9	REQUIRES and EXCLUDES	CTC_0: $A \to B$	$Rt_0Et_0 = \mathcal{T}(+B-C)$	✓	CTC_0: $B \to \neg C$	$Et_0Rt_0 = \mathcal{T}(-C+B)$	✓	First the constraint with the shared feature in the consequent (i.e., the requires constraint). Follow order 1.
			CTC_1: $B \to \neg C$	$Rt_0Et_1 = \mathcal{T}(+B-B+C)$	✗	CTC_1: $A \to B$	$Et_0Rt_1 = \mathcal{T}(-C-A-B)$	✓	
				$Rt_1Et_0 = \mathcal{T}(-A-B-C)$	✓		$Et_1Rt_0 = \mathcal{T}(-B+C+B)$	✗	
				$Rt_1Et_1 = \mathcal{T}(-A-B-B+C)$	✓		$Et_1Rt_1 = \mathcal{T}(-B+C-A-B)$	✓	
	C10	REQUIRES and EXCLUDES	CTC_0: $B \to A$	$Rt_0Et_0 = \mathcal{T}(+A-B)$	✓	CTC_0: $C \to \neg B$	$Et_0Rt_0 = \mathcal{T}(-B+A)$	✓	First the constraint with the shared feature in the consequent (i.e., the excludes constraint). Follow order 2.
			CTC_1: $C \to \neg B$	$Rt_0Et_1 = \mathcal{T}(+A-C+B)$	✓	CTC_1: $B \to A$	$Et_0Rt_1 = \mathcal{T}(-B-B-A)$	✓	
				$Rt_1Et_0 = \mathcal{T}(-B-A-B)$	✓		$Et_1Rt_0 = \mathcal{T}(-C+B+A)$	✓	
				$Rt_1Et_1 = \mathcal{T}(-B-A-C+B)$	✗		$Et_1Rt_1 = \mathcal{T}(-C+B-B-A)$	✗	

* ✓: The transformations result in a valid tree; ✗: The transformations result in an empty tree (NIL).

for two simple constraints considering all possibilities for well-formed valid constraints. The analysis is divided first by whether the shared feature across the constraints appears in the antecedent or consequent of the constraint (column `Rule`). Then, we consider the constraint type: *requires*, *excludes*, or both (column `Type`). Finally, we consider the two possible orders for the two constraints (Columns `Order 1` and `Order 2`), giving 10 different cases (`C1` to `C10`). For each case, we show the four instances of the transformations vector with the four possible subtrees of $\mathcal{T}$ (column `Transformations`). The `Valid` column indicates whether the transformations result in a valid or an empty subtree (NIL).

We obtain the following four insights regarding the order of the constraints in the transformations vector. (1) It should first put the constraints that share a feature in which the feature appears in both constraints in the consequent of the implication regardless of the constraint type (cases `C4`, `C5`, and `C6`). (2) Second, the constraints that share a feature, in which the feature appears in the antecedent in one constraint and in the consequent in the other constraint (cases `C7` to `C10`). At this point, it is preferable to sort the constraints by putting the constraint in which the shared feature appears in the consequent first. It is because, in these cases, the transformations vector reaches an empty tree earlier (`Order 1` in cases `C7`, `C8`, and `C9`, and `Order 2` in case `C10`). (3) Finally, the constraints that share a feature in the antecedent and the rest that do not share any features. (4) Another conclusion is that sorting the constraints by type (*requires* or *excludes*) is irrelevant because it always results in the same number of empty subtrees.

Given an order of constraints, $\mathcal{TV} = C_0, C_1, \ldots, C_{n-1}$, the total number of empty subtrees can be calculated using Algorithm 3. This algorithm iterates through the constraints (lines 4 to 11), evaluating the number of empty subtrees by pairs (line 5). The number of empty subtrees by pairs is calculated using the function $R_{NILS}(C_l, C_r)$. This function returns the number of empty subtrees that are obtained when the C_l constraint is applied before the C_r one. When a pair returns empty subtrees (lines 7 to 9), it calculates the total number of empty subtrees that corresponds with the number of instances of the vector that will be skipped from execution. This number depends on the position (i) of the constraint in the ordering (line 9). The Algorithm 3 can be extended to more constraint combinations (three, four, ...).

If a model has n constraints, $n!$ are all possible ordering. This vast space cannot be explored by brute force. For instance, in a medium size model like JHipster, this number is up to 1,196e56. Even if we can calculate Algorithm 3 each nanosecond, it will take more than

Algorithm 3 Fitness function for two constraints in H_4 (FSCH).

Input: $\mathcal{TV}$: The transformations vector with a constraints ordering $C_0, C_1, \ldots, C_{n-1}$.
Output: Number of empty subtrees.

1: **function** FITNESS_VALID($\mathcal{TV}$)
2: $fitn \leftarrow 0$
3: $i \leftarrow 0$
4: **while** $i < n$ **do**
5: **if** $R_{NILS}(C_i, C_{i+1}) = 0$ **then**
6: $i \leftarrow i + 1$
7: **else**
8: $i \leftarrow i + 2$
9: $fitn \leftarrow fitn + R_{NILS}(C_i, C_{i+1}) * 2^{n-i}$ ⊳ Order matters.
10: **end if**
11: **end while**
12: **return** $fitn$
13: **end function**

3e39 years to complete. Evolutionary algorithms can find solutions in huge search spaces. They evolve individuals (e.g., a possible ordering of constraints) over generations to improve their fitness (an associated value for each individual, Algorithm 3). We have used the $\mu + \lambda$ evolutionary algorithm implemented in DEAP (Fortin et al., 2012) with the Algorithm 3 with 2, 3 and 4 constraints as the fitness function (Section 7).

5. FM$_{\text{Sans}}$: A constraints removal-based knowledge compilation technique

This section details step 3 of our approach where we construct the FM$_{\text{Sans}}$ model.

The left-hand side of Fig. 6 illustrates our knowledge compilation technique, called FM$_{\text{Sans}}$. It consists of three main elements: (1) a *feature tree* as defined for FM$_{\text{relaxed}}$ that, according to Definition 5, may have auxiliary abstract features; (2) a *Transformations vector* $\mathcal{TV}$ that encodes the transformations to be performed to eliminate each constraint according to Definition 7; and (3) a set of *ids* that identify the instances of the transformations vector that result in valid subtrees. Let us formally define our knowledge compilation technique:

Definition 8 (*FM$_{Sans}$*). A FM$_{\text{Sans}}$ is a knowledge compilation for feature models defined as a 3-tuple $(\mathcal{T}', \mathcal{TV}, \mathcal{ST}_{ids})$, where:

- $\mathcal{T}'$ is a feature tree following Definition 5.
- $\mathcal{TV}$ is the transformations vector following Definition 7.
- $\mathcal{ST}_{ids}$ is a set of natural numbers $x \in \mathbb{N}$ identifying the binary values for which the execution of the instance v_x of the transformations vector $\mathcal{TV}$ returns a valid (non-empty) subtree of $\mathcal{T}'$. The integer numbers hold $\forall x \in \mathcal{ST}_{ids}, 0 \leq x \leq 2^n - 1$. ■

For completeness, a feature model without constraints is codified in an FM$_{\text{Sans}}$ as $(\mathcal{T}', [], \emptyset)$ representing a unique, valid feature tree with an empty transformations vector ($\mathcal{TV}$) and an empty set of valid subtrees identifiers ($\mathcal{ST}_{ids}$). The bottom-left-hand side of Fig. 6 shows the FM$_{\text{Sans}}$ for the GPL running example. It contains the feature tree $\mathcal{T}'$ from the FM$_{\text{relaxed}}$ and encodes the five simple constraints in the transformations vector $\mathcal{TV}$. The set of valid ids $\mathcal{ST}_{ids}$ contains 14 identifiers that correspond with those instances of the transformations vector $\mathcal{TV}$ which results in valid subtrees (see right-hand side of Fig. 6). Initially, $\mathcal{ST}_{ids}$ is empty, and the ids are obtained as follows.

Naive approach to identifying valid subtrees. A naive approach to identifying the valid subtrees is to execute all possible binary vectors $v_x \forall x, 0 \leq x \leq 2^n - 1$, in $\mathcal{TV}$. However, it is easy to note that this approach will not scale even for a small number of constraints (e.g., 20 constraints imply executing more than 1 million vectors). We propose a better approach considering the independence of each vector v_x and the binary encoding.

Parallel approach to identifying valid subtrees. The binary encoding of the transformations vector $\mathcal{TV}$ allows us to split the search space into independent parts for parallelization (top-left of Fig. 6). First, we use the initial T bits of the transformations vector $\mathcal{TV}$ to divide the work into 2^T independent tasks that can be distributed on different computers. For instance, we can use the first bit to define two tasks: executing all vectors starting with 0 and executing all vectors starting with 1. Second, we use the next C bits to divide the remaining vectors into 2^C processes to exploit multiprocessing on individual multi-core computers. In our running example (right-hand side of Fig. 6), we split the work into two distributed tasks, and each task is divided into 4 processes. Each process only needs to consider the remaining $n - T - C$ bits of each binary vector.

Pruning approach to skip invalid (empty) subtrees. In addition, in each parallel process, during the execution of the vectors, we apply a pruning approach so that when the transformation associated with the bit i results in an empty subtree (NIL), the remaining bits to the right in the $\mathcal{TV}$ do not need to be executed regardless of their values. Thus, as explained in Section 4.2 we skip the execution of $2^{n-i} - 1$ vectors.

At the end of the process, we collect all valid identifiers of those binary vectors that result in non-empty subtrees to form the $\mathcal{ST}_{ids}$ of our FM$_{\text{Sans}}$. For the GPL excerpt, we obtain 14 valid subtrees from 32 possible binary vectors, of which we only executed 24 (8 have been skipped). The FM$_{\text{Sans}}$ is serialized in an external file in *.json* format for further analyses that includes its three main concepts: the feature tree $\mathcal{T}'$, the transformations vector codification $\mathcal{TV}$, and the set of valid subtrees ids $\mathcal{ST}_{ids}$.

6. Automated analysis of FM$_{\text{Sans}}$

Once the FM$_{\text{Sans}}$ model has been constructed, we can analyze it in parallel (steps 4 to 6 in Fig. 4) or obtain a generalized feature tree (GFT) (steps 7 and 8 in Fig. 4).

6.1. Parallel analysis of valid subtrees

To analyze an FM$_{\text{Sans}}$, we first generate each subtree $\mathcal{T}_x$ associated with its valid identifier $x \in \mathcal{ST}_{ids}$. Each subtree $\mathcal{T}_x$ is generated by using the feature tree $\mathcal{T}'$ and applying the binary vector $v_x \in \mathcal{TV}$ to $\mathcal{T}'$. We obtain $\mathcal{T}_x$ using the same method we used to obtain its valid identifier in the previous section (step 3 of our approach). After executing all transformations represented by the vector v_x, we obtain a subtree $\mathcal{T}'_x$ that may still contain auxiliary abstract features ($\mathcal{F}_{aux}$), in case the input was an FM$_{\text{relaxed}}$. These auxiliary abstract features were introduced to remove the complex constraints, and with the removal of the simple constraints, those features are no longer needed. The reason is that they are leaf abstract features not mapped to software artifacts and must be removed to maintain the semantics of the original FM. As a result, we obtain a final subtree $\mathcal{T}_x$ as in Definition 2 without auxiliary abstract features. The subtree $\mathcal{T}_x$ represents a disjoint subset of the products of the original FM.

At this point, we can use various AAFM techniques, such as SAT solving or BDD, to analyze the subtree as a regular feature model without constraints. Here, we propose to analyze the subtree by applying recursive analysis operations over the hierarchical structure of the subtree (see Algorithms 4 to 8). Both the generation (step 4) and analysis of each subtree (step 5) are executed by exploiting process-based parallelism, where a pool of processes is spawned for the set of valid subtrees in $\mathcal{ST}_{ids}$. After the analysis, each subtree is discarded to free up memory, and the analysis results are joined (step 6) to get the final results of the FM$_{\text{Sans}}$ analysis.

We define the following well-known analysis operations (Benavides et al., 2010) for FM$_{\text{Sans}}$ models. For each analysis operation, we define two functions: (i) a joint function in charge of joining the results of all subtrees in the FM$_{\text{Sans}}$; and (ii) the function that computes the analysis over a single feature (sub)tree adapted from van den Broek and Galvão (2009):

- **Valid** (Algorithm 4). A FM$_{\text{Sans}}$ is valid if it represents at least one product. A non-empty feature tree $\mathcal{T}$ is always valid since there are no constraints. Therefore, a FM$_{\text{Sans}}$ is valid if it has at least one valid subtree in $\mathcal{ST}_{ids}$ or its original feature tree $\mathcal{T}$ is non-empty (in case of empty $\mathcal{ST}_{ids}$ due to no constraints codification).
- **Number of products** (Algorithm 5). The products' number of an FM$_{\text{Sans}}$ is the sum of the products' number of all subtrees in $\mathcal{ST}_{ids}$. The products' number of a feature tree $\mathcal{T}$ is calculated by recursively traversing the tree structure of $\mathcal{T}$ from the root to the leaves considering the relation type of each feature.

Algorithm 4 Valid.

Input: fm: The FM_{Sans} model.
Output: *True* if the fm represents at least one product; *False* otherwise.
1: **function** Join_Valid(fm)
2: **return** $fm.\mathcal{ST}_{ids} \neq \emptyset \vee$ Valid($fm.\mathcal{T}'$)
3: **end function**
Input: $\mathcal{T}$: The feature tree.
Output: *True* if the feature tree represents at least one product; *False* otherwise.
4: **function** Valid($\mathcal{T}$)
5: **return** $\mathcal{T} \neq NIL$ ▷ A feature tree has products if and only if it is not empty.
6: **end function**

Algorithm 5 Number of products.

Input: fm: The FM_{Sans} model.
Output: The number of products of the fm model.
1: **function** Join_ProductsNumber(fm)
2: **return** $\sum$ ProductsNumber($\mathcal{T}$, $\mathcal{T}.r$), $\forall \mathcal{T} \in fm.\mathcal{ST}_{ids}$ ▷ Sum of the products' number.
3: **end function**
Input: $\mathcal{T}$: The feature tree, f: The root feature ($\mathcal{T}.r$).
Output: The number of products represented by the feature tree.
4: **function** ProductsNumber($\mathcal{T}$, f)
5: **if** $is_leaf(f)$ **then return** 1 ▷ The feature f has no children.
6: **else**
7: $n \leftarrow 1$
8: **for all** $g \in \mathcal{T}.\mathcal{F} | \exists relation(f,g) \in \mathcal{T}.\mathcal{R}$ **do**
9: **if** is_mandatory(g) **then** $n \leftarrow n *$ ProductsNumber($\mathcal{T}$, g)
10: **else if** is_optional(g) **then** $n \leftarrow n * (1+$ ProductsNumber($\mathcal{T}$, g))
11: **else if** is_xor_group(f) **then** $n \leftarrow n * (\sum$ ProductsNumber($\mathcal{T}$, g))
12: **else if** is_or_group(f) **then** $n \leftarrow n * (\prod(1+$ ProductsNumber($\mathcal{T}$, g))-1)
13: **end if**
14: **end for**
15: **end if**
16: **return** n
17: **end function**

- **Configurations** (Algorithm 6). The configurations of an FM_{Sans} are the union of the configurations of each feature subtree in $\mathcal{ST}_{ids}$ (see `Join_Configuration` function in lines 1–7). Similar to the number of products, the configurations of a feature tree $\mathcal{T}$ are calculated by recursively traversing the tree structure of $\mathcal{T}$ from the root to the leaves, considering the relation type of each feature (see line 17 Algorithm 6), and carrying the set of configurations from the leaves to the root feature combining the set of configurations according to the relation type (see lines 18–26). The number of products of FM_{Sans} and the number of configurations of FM_{Sans} are equal because the configurations only contain abstract features in the internal nodes of the tree. The products can be obtained directly by filtering the configurations to maintain only the concrete features.
- **Core features** (Algorithm 7). The core features of an FM_{Sans} is the intersection of the core features of all subtrees in $\mathcal{ST}_{ids}$. The core features of a feature tree $\mathcal{T}$ are the root feature and all its mandatory children recursively.
- **Dead features** (Algorithm 8). The dead features of an FM_{Sans} are those features in the original feature tree $\mathcal{T}'$ (without auxiliary abstract features) that are not present in any subtree in $\mathcal{ST}_{ids}$. A feature tree $\mathcal{T}$ has no dead features since the dead features are caused by constraints.

In all cases, if FM_{Sans} does not encode any constraints (i.e., $\mathcal{TV}$ is empty), the result of each analysis operation is the result of applying them to the original feature tree $\mathcal{T}'$.

Other operations, such as commonalities (Fernández-Amorós et al., 2014), feature cardinalities (Heradio et al., 2019; Sundermann et al., 2021b) or the analysis of partial configurations can be implemented in our approach by forcing the presence of a particular feature in the subtrees (i.e., using the `Commitment` operation defined in Algorithm 1).

Algorithm 6 Configurations.

Input: fm: The FM_{Sans} model.
Output: The configurations of the fm model.
1: **function** Join_Configurations(fm)
2: **if** $fm.\mathcal{ST}_{ids} = \emptyset$ **then**
3: **return** Configurations($fm.\mathcal{T}$, $fm.\mathcal{T}.r$)
4: **else**
5: **return** $\bigcup$ Configurations($\mathcal{T}$, $\mathcal{T}.r$), $\forall \mathcal{T} \in fm.\mathcal{ST}_{ids}$ ▷ Union of configurations.
6: **end if**
7: **end function**
Input: $\mathcal{T}$: The feature tree, f: The root feature ($\mathcal{T}.r$).
Output: The set of configurations of the feature tree.
8: **function** Configurations($\mathcal{T}$, f)
9: $config \leftarrow \{f\}$
10: **if** $is_leaf(f)$ **then**
11: **return** $\{config\}$
12: **end if**
13: $all_configs \leftarrow \emptyset$
14: $xor_configs \leftarrow \emptyset$
15: $or_configs \leftarrow \emptyset$
16: **for all** $g \in \mathcal{T}.\mathcal{F} | \exists relation(f,g) \in \mathcal{T}.\mathcal{R}$ **do**
17: $sub_configs \leftarrow \{c \cup \{f\}, \forall c \in$ Configurations($\mathcal{T}$, g)$\}$
18: **if** is_mandatory(g) **then**
19: $all_configs \leftarrow \{c \cup s, \forall c \in all_configs, \forall s \in sub_configs\}$
20: **else if** is_optional(g) **then**
21: $all_configs \leftarrow all_configs \cup \{c \cup s, \forall c \in all_config, \forall s \in sub_configs\}$
22: **else if** is_xor_group(f) **then**
23: $xor_configs \leftarrow xor_configs \cup \{sub_configs\}$
24: **else if** is_or_group(f) **then**
25: $or_configs \leftarrow or_configs \cup \{sub_configs\}$
26: **end if**
27: **end for**
▷ Add the alternative configurations in the xor-group.
28: $all_configs \leftarrow \{c \cup s, \forall c \in all_config, \forall s \in X\}, \forall X \in xor_configs$
▷ Add the combinations of configurations taken from 1 to n in the or-group.
29: $all_configs \leftarrow \{c \cup s, \forall c \in all_config, \forall s \in X\}, \forall X \in \bigcup_{i=1}^{|or_config|} \{c \in \binom{or_configs}{i}\}$
30: **return** $all_configs$
31: **end function**

Algorithm 7 Core features.

Input: fm: The FM_{Sans} model.
Output: The core features of the fm model.
1: **function** Join_CoreFeatures(fm)
2: **return** $\bigcap$ CoreFeatures($\mathcal{T}$, $\mathcal{T}.r$), $\forall \mathcal{T} \in fm.\mathcal{ST}_{ids}$ ▷ Intersection of core features.
3: **end function**
Input: $\mathcal{T}$: The feature tree, f: The root feature ($\mathcal{T}.r$).
Output: The core features of the feature tree.
4: **function** CoreFeatures($\mathcal{T}$, f)
5: **return** $\bigcup$ CoreFeatures($\mathcal{T}$, g), $\forall g \in \mathcal{T}.\mathcal{F} | \exists$relation($f$,$g$) $\in \mathcal{T}.\mathcal{R} \wedge$ is_mandatory(g)
6: **end function**

Algorithm 8 Dead features.

Input: fm: The FM_{Sans} model.
Output: The dead features of the fm model.
1: **function** Join_DeadFeatures(fm)
2: **if** $fm.\mathcal{ST}_{ids} = \emptyset$ **then return** $\emptyset$
3: **else return** $fm.\mathcal{T}'.\mathcal{F} \setminus (\bigcup \mathcal{T}.\mathcal{F}, \forall \mathcal{T} \in fm.\mathcal{ST}_{ids})$ ▷ Features that are not in subtrees.
4: **end if**
5: **end function**
Input: $\mathcal{T}$: The feature tree.
Output: The dead features of the feature tree.
6: **function** DeadFeatures($\mathcal{T}$)
7: **return** $\emptyset$ ▷ A feature tree has not dead features.
8: **end function**

6.2. Generation and analysis of the GFT

An additional step of our approach is that we can compound a GFT (step 7) from the FM_{Sans} model by joining all subtrees after generating them in step 4. This is done by incorporating an auxiliary abstract root feature (as in Fig. 3), which is an alternative group, and its children are the root features of each subtree in $\mathcal{ST}_{ids}$. Therefore, the subtrees still have disjoint semantics since they belong to different branches of an alternative (xor) group. Generating the GFT requires maintaining all subtrees in memory, which can be unsuitable when the number of

subtrees is considerable (see Section 7). The GFT can be serialized in the Universal Variability Language (UVL) (Benavides et al., 2025) format since it supports specifying duplicate features (Benavides et al., 2025). Finally, the GFT can also be analyzed (step 8 of our approach) using the same analysis operations presented in Algorithms 4 to 8 for feature (sub)trees. The advantage of analyzing the GFT is that the complexity of the analysis operations is linear in the size of the GFT, but the drawback is that the size is exponential in the number of constraints (see next Section).

7. Evaluation

We implemented our approach and conducted experiments to evaluate the following research questions. The artifacts, including the feature models in all versions presented through the paper and information on how to replicate the evaluation are available online.[2]

RQ1 *To what extent do feature models increase in size by removing all constraints?* **Rationale:** The main problem with representing constraints in the feature tree is the exponential increase in the number of features and the presence of duplicate features. The goal is to study the increase in feature models' size in our approach compared to the original model, the $FM_{relaxed}$ and the GFT.

RQ2 *How does the order of the constraints affect the performance of removing the constraints?* **Rationale:** The order in which the constraints are codified in the transformations vector affects the number of instances of the vector that need to be executed to obtain all valid subtrees. The aim is to check whether our heuristics are able to reduce the number of transformations to be executed.

RQ3 *What are the limits for a solver based on the tree hierarchy structure of the feature tree?* **Rationale:** Analysis operations over a GFT can be very efficient since no translation to SAT or BDD is required. However, the number of features in the GFT can be colossal (i.e., millions of features) even for medium-size models, making it harder to manage the model itself. The goal is to study if our approach improves the analysis of the models and, thus, the FM_{Sans} can be used with those models that cannot be handled with a fully assembled GFT due to its size.

RQ4 *To what extent do parallelization techniques improve the automated analysis of feature trees?* **Rationale:** Moving the complexity from memory to time can be counter-productive for some analyses. To alleviate the inherent exponential complexity in time, our approach enables distribution and parallelization. The aim is to check whether our parallelization approach improves the identification and analysis of the subtrees.

7.1. Open-source implementation

We provide an implementation of our approach in Python, built on top of *Flamapy*,[3] a Python framework for AAFMs (Galindo et al., 2023). Our implementation includes a port in Python of Knüppel's algorithm (Knüppel et al., 2017) to eliminate complex constraints initially available in Java for FeatureIDE (Thüm et al., 2014b), as well as an adaptation of Broek's algorithms (van den Broek and Galvão, 2009; van den Broek et al., 2008) to eliminate simple constraints and to analyze the GFT originally developed in Miranda (Turner, 1985). In addition, we propose an implementation of the analysis operations presented in Section 6 over FM_{Sans} models.

[2] Artifact: https://github.com/CAOSD-group/fmsans.
[3] *Flamapy*: https://flamapy.github.io/.

7.2. Experimental setup and models corpus

Feature models. We consider up to 23 feature models (Table 2) in UVL (Benavides et al., 2025) format varying in size and complexity, including real-world standard benchmark models used in the SPL community (Sundermann et al., 2024; Åkesson et al., 2019; Ghamizi et al., 2019; Halin et al., 2019; Horcas, 2018; Horcas et al., 2022; Kowal et al., 2016; Krieter, 2020; Martinez et al., 2018; Oh et al., 2019; Pett et al., 2021; Schmitt et al., 2018; She et al., 2010; Siegmund et al., 2012; Tieber and Felfernig, 2021; Young, 2005). For EMB ToolKit and Linux models, whose relaxed FMs cannot be obtained with Flamapy, we use the models available in Knüppel's repository[4] obtained using the original algorithm with FeatureIDE (Knüppel et al., 2017). The models are ordered by the number of simple constraints in its relaxed version. We classify the models in three groups according to the number of constraints: (1) small-size models (up to 40 constraints); (2) medium-size models (from 40 to 200 constraints); and (3) large-size models (more than 200 constraints). Small models serve as testing models to guarantee correctness and validate our approach (e.g., the Tank War model without constraints is useful as the boundary base case). Medium-size models are useful because, as small-size models, they can be manually debugged. Large-size models pose a challenge in scalability and serve to analyze the limits of our approach.

Algorithms and parameter settings. We use SAT solvers and BDDs to analyze feature models and compare the results with our analysis operations over the FM_{Sans}. To work with SAT and BDD in Python we rely on Flamapy. Concretely, for SAT, we use the *PySAT* library (Ignatiev et al., 2018) with the *Glucose3* (Audemard and Simon, 2018) solver to double-check the analysis results. For BDD, we use the *BDD4va*[5] library, a wrapper of the *BDDSampler* artifact proposed in Heradio et al. (2022) to compare the efficiency of the analysis operations. For the genetic algorithm used in the FSCH heuristic, we use a population of 100 individuals with 50 generations, the rest of the parameters are the defaults one in $\mu + \lambda$ evolutionary algorithm in DEAP (Fortin et al., 2012): a mutation rate of 0.2 and a probability that an offspring is produced by crossover of 0.7. A deep analysis with different parameters for the genetic algorithm is out of scope of this paper.

Execution setup. For small and medium-size models with less than 70 constraints, we perform 30 runs and calculate the medians, means, and standard deviations for execution time. We execute only one run with a timeout of 24 h for large-size models and medium models with more than 70 constraints.

To compare the heuristics, we use the medians and maximum values of the execution time and the number of analyzed feature trees in the process of identifying the valid feature trees. We have also applied the Mann–Whitney U test (Arcuri and Briand, 2014) for statistical analysis (Table 6), commonly used with randomized algorithms in software engineering, to check whether the differences between the heuristics are significant. We report p-values, where a value below 0.05 means that the comparison is statistically significant at the 95% confidence level.

The experiments were performed on a desktop computer with Intel Core i5-10400 CPU @ 3.6 GHz x 6, 16 GB RAM, Linux Mint 21.1, and Python 3.10. Additionally, for large scale models, we perform the experiments to build the FM_{Sans} on a supercomputer[6] with 128 processes with a timeout of 24 h.

[4] Relaxed feature models: https://github.com/AlexanderKnueppel/is-there-a-mismatch.
[5] *BDD4va*: https://github.com/rheradio/bdd4va.
[6] *Picasso* supercomputer: https://www.scbi.uma.es/site/.

Table 2

Comparison of the original feature models (FM), the relaxed feature models ($FM_{relaxed}$), the subtrees of our approach (FM_{Sans}), and the generalized feature trees (GFT), in terms of the feature tree, constraints, and semantics.

	Original feature model (FM)							Relaxed feature model ($FM_{relaxed}$)						Our approach (FM_{Sans})							GFT	
	Features	Constraints					Semantics	Features		Constraints			Semantics	Subtrees statistics						Semantics	Features	Semantics
Feature model	$\mathcal{F}$	Ψ	Ψ_{s_R}	Ψ_{s_E}	Ψ_{c_P}	Ψ_{c_S}	$\mathcal{PR} = \mathcal{CF}$	$\mathcal{F}$	$\mathcal{F}_{aux}$	Ψ_s	Ψ_{s_R}	Ψ_{s_E}	$\mathcal{CF}$[1]	$\mathcal{ST}$	$\mathcal{F}_{min}$	$\mathcal{F}_{max}$	$\mathcal{F}_{\eta}$	$\mathcal{F}_{\mu}$	$\mathcal{F}_{\sigma}$	$\mathcal{PR} = \mathcal{CF}$	$\mathcal{F}$	$\mathcal{PR} = \mathcal{CF}$
Tank War Martinez et al. (2018)	37	0	0	0	0	0	2e6	37	0	0	0	0	2e6	1	37	37	37	37	0	2e6	37	2e6
Mobile Media Young (2005)	43	3	3	0	0	0	2e6	43	0	3	3	0	2e6	8	34	43	39	38	3.0	2e6	303	2e6
Pizzas Knüppel et al. (2017)	12	2	0	0	1	1	25	16	4	5	1	4	49	16	7	10	8	8	0.9	25	131	25
GPL excerpt	12	2	0	0	1	1	36	16	4	5	3	2	104	14	7	10	8	8	0.9	36	117	36
WeaFQAs Horcas (2018)	179	7	6	1	0	0	3e24	179	0	7	6	1	3e24	128	131	178	155	154	13.6	3e24	19,713	3e24
Truck Schmitt et al. (2018)	33	10	8	2	0	0	234	33	0	10	8	2	234	42	14	21	18	18	1.9	234	745	234
Apo-Games Åkesson et al. (2019)	63	14	12	0	0	2	4e8	72	9	19	17	2	8e8	7	43	48	47	47	1.7	4e8	328	4e8
MTG Cards Tieber and Felfernig (2021)	781	18	10	0	1	7	7e214	812	31	41	28	13	5e216	163,728	355	759	436	494	131.0	7e214	8e7	7e214
JHipster Halin et al. (2019)	45	13	2	1	1	9	26,256	94	49	44	34	10	2e7	79	14	31	28	26	4.8	26,256	2027	26,256
BerkeleyDB Siegmund et al. (2012)	76	20	10	0	10	0	4e9	76	0	46	46	0	4e9	656	35	76	44	44	4.7	4e9	28,897	4e9
Graphs VIZ Horcas et al. (2022)	86	19	0	2	0	17	37,706	162	76	61	33	28	6.7e10	310	11	22	17	17	2.2	37,706	5134	37,706
BerkeleyDB C Oh et al. (2019)	19	29	0	0	0	29	3,840	107	88	61	61	0	2e17	30	17	19	17	17	0.6	3,840	522	3,840
Financial Ser. Krieter (2020)	580	61	33	11	0	17	7e101	628	48	76	65	11	3e105	14% binary vectors explored in 24 hours							unknown	unknown
GPL Lopez-Herrejon and Batory (2001)	66	32	4	0	6	22	107,094	156	90	86	40	46	3e14	94,770	7	41	29	29	4.3	107,094	3e6	107,094
DNN archit. Ghamizi et al. (2019)	1,761	110	65	45	0	0	5e162	1,761	0	110	65	45	5e162	9,330	17	1,716	837	832	266.7	5e162	8e6	5e162
axTLS Knüppel et al. (2017)	96	14	1	0	3	10	8e11	316	220	169	79	90	5e24	432	63	83	71	72	4.0	8e11	30,925	8e11
Busy Box Pett et al. (2021)	631	681	527	36	0	118	4e141	1,108	477	930	755	175	timeout[3]	0.4% binary vectors explored in 24 hours							unknown	unknown
uClinux distr. Knüppel et al. (2017)	1,580	197	101	2	52	42	4e409	2,601	1,021	1,044	668	376	timeout[3]	56% binary vectors explored in 24 hours							unknown	unknown
Automotive 2.4 Kowal et al. (2016)	18,616	1,369	1,005	55	94	215	6e1543	19,857	1,258	2,394	1,848	546	timeout[3]	6e-9% binary vectors explored in 24 hours							unknown	unknown
uClinux-base Knüppel et al. (2017)	380	3,455	38	3,384	0	33	3e22	704	324	3,713	299	3,414	3e36	93% binary vectors explored in 24 hours							unknown	unknown
CDL ea2468 Knüppel et al. (2017)	1,408	956	685	29	123	119	timeout[3]	9,435	8,139	7,795	3,916	3,879	timeout[3]	6e-3% binary vectors explored in 24 hours							unknown	unknown
EMB ToolKit Knüppel et al. (2017)	1,179	323	67	2	timeout[2]		unknown	17,857	16,678	15,382	13,314	2,068	timeout[3]	95% binary vectors explored in 24 hours							unknown	unknown
Linux 2.6.33.3 She et al. (2010)	6,467	3,545	184	62	memoryout[4]		unknown	45,141	38,674	29,175	18,403	10772	timeout[3]	73% binary vectors explored in 24 hours							unknown	unknown

$\mathcal{F}$: Features, $\mathcal{F}_{aux}$: Auxiliary abstract features, Ψ: Constraints, Ψ_s: Simple constraints, Ψ_{s_R}: Requires constraints, Ψ_{s_E}: Excludes constraints, Ψ_{c_P}: Pseudo-complex constraints, Ψ_{c_S}: Strict-complex constraints, $\mathcal{PR}$: Number of products, $\mathcal{CF}$: Number of configurations, $\mathcal{ST}$: Subtrees, $\mathcal{F}_{min}$: Minimum number of features in subtrees, $\mathcal{F}_{max}$: Maximum number of features in subtrees, $\mathcal{F}_{\eta}$: Median of features in subtrees, $\mathcal{F}_{\mu}$: Mean of features in subtrees, $\mathcal{F}_{\sigma}$: Standard deviation of features in subtrees.

[1] The configurations ($\mathcal{CF}$) and products ($\mathcal{PR}$) differ only for those relaxed feature models with auxiliary abstract features ($\mathcal{F}_{aux}$), but the models also maintain the semantics ($\mathcal{PR}$) according to Knüppel et al. (2017).

[2] Timeout of 24 h processing complex constraints. [3] Timeout of 24 h building the BDD. [4] Memory blow-up processing complex constraints.

Highlighted in gray the number of simple constraints used as baseline for size comparison in terms of constraints and used for ordering the feature models.

In yellow relaxed feature models obtained from (Knüppel et al., 2017) using the original Knüppel's algorithm in FeatureIDE. In red the generalized feature trees that cannot be serialized due to their sizes.

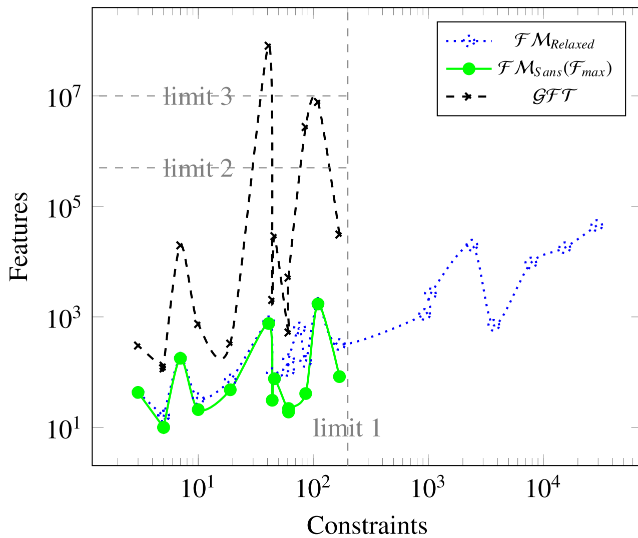


Fig. 7. Comparison of the feature models' size in terms of features when removing constraints *w.r.t.* $\text{FM}_{\text{relaxed}}$.

7.3. Results and discussion

RQ1. To what extent do feature models increase in size by removing all constraints? We compare the size of the feature models (Tables 2 and 3) in terms of the number of features and constraints for: (i) the original feature model (FM); (ii) the $\text{FM}_{\text{relaxed}}$ with only simple constraints (obtained in step 1 of our approach); (iii) the subtrees of the FM_{Sans} (obtained in step 3); and (iv) the GFT with no constraints (generated in step 7). So far, we have proved that in all models, the semantics is maintained (column $\mathcal{PR}$ in Table 2).

Knüppel et al. (2017) demonstrated that for all the models with complex constraints that they used in their evaluation, the $\text{FM}_{\text{relaxed}}$ increased in size from below 1% to 1,403% in features and from 7% to 4,648% in constraints (Knüppel et al., 2017). This explosion in size is even worse when simple constraints are removed (column GFT in Table 3), preventing the application of AAFM techniques to large-scale models. For instance, 6 of 7 of the large models lead to a timeout when building the BDD for its $\text{FM}_{\text{relaxed}}$ (see Table 2). Concretely, CDL ea2468, EMB ToolKit, and Linux models are particularly inefficient when processed (e.g., reading, traversing, extracting properties) even in their original format, as shown in Table 2, with several timeouts and memory blow-ups while processing complex constraints.

Fig. 7 shows how our approach does not increase the size of the models.[7] This is because the larger subtree generated has, in the worst case, the same size (in number of features) as the relaxed feature model (see also the percentage of increments in Table 3). Regarding the size of the GFT, despite its size increasing exponentially in the number of simple constraints, our approach allows analyzing the models without generating the complete GFT. In this respect, although the GFT can be built for medium-size models such as GPL or DDN architectures, the models contain millions of features that prevent them from being serialized in a UVL file. The MTG Cards model cannot even be built in memory because its GFT has over 80 million features (see memory blow-up in Table 3). Using our approach, these models can be analyzed in parallel using the same analysis operations used for their equivalent GFT but applying them to each smaller subtree of the FM_{Sans}.

RQ2. How does the order of the constraints affect the performance of removing the constraints? We study how four different orderings of the constraints (step 2 in our approach) affect the performance of obtaining the valid subtrees (step 3).

Tables 4 and 5 show the results of applying our heuristics to reorder the constraints in the transformations vector TV and to identify the valid subtrees (see Table 4 for small and medium-size models, and Table 5 for large-size models). Table 6 shows the percentages of improvements after applying the heuristics compared to the normal ordering. The number of subtrees analyzed is the number of instances of the transformations vector TV executed, resulting in both valid subtrees and empty subtrees. Note that for a given feature model, the number of valid subtrees is always the same since the division of the configuration space into disjoint trees is always equal. The only difference lies in the process of identifying those valid subtrees. For each model, the significant difference between the final number of valid subtrees and the total number of possible transformation vectors executed to identify them (2^n being n the number of constraints) is worth noting. On the one hand, for models with more than 20 constraints, the percentage of valid subtrees is minuscule (e.g.,0.001% for the `Apo-Games` model with 19 constraints, 7.44e−6% for the `GPL` model with 86 constraints). This is mainly due to the configuration spaces being heavily restricted by the constraints imposed in the original model compared to the number of features they possess. The opposite circumstance can be observed in models with more features than constraints (`Mobile Media` and `WeaFQAs` where all possible subtrees are valid).

The *ratio* column in Tables 4 and 5 is the division between the number of pruned subtrees (i.e., skipped executions) and the number of analyzed subtrees (i.e., valid subtrees plus empty subtrees whose transformations vector has been executed). The *ratio* allows the comparison of the heuristics using a dimensionless unit. The ratio indicates the pruning speed in identifying valid subtrees, and it can be observed that the heuristics scale well for medium-size models where the space is very scattered. In general, it is observed that the heuristic based on tree size (FTSH) achieves better results for most models, as it analyzes a smaller number of subtrees. For example, for the `GPL` feature model, with FTSH, 54.79% fewer subtrees are analyzed than with the normal order (NOH), resulting in a time improvement of 39.17% (Table 6). This is because FTSH uses model-specific information, such as the number of features to be removed from the tree with each constraint. However, the genetic heuristic based on shared features in constraints (FSCH) achieves better results in models with many constraints that share features. The reason is that removing these constraints has a more significant effect on the tree (models `Truck`, `MTG Cards`, and `BerkeleyDB`). For example, in the `BerkeleyDB` model, heuristic FSCH reduces the number of analyzed trees by 89.69% and the process time by more than half (56.49%). There are some cases where the heuristic reduces the number of analyzed trees, but the execution time is higher (e.g., FTSH in `JHipster` and `Berkeley C`). This indicates that the cost of removing each constraint is different. Additionally, the first constraints in the transformations vector operate on the complete feature tree and its size decreases as constraints are removed, so even though more transformations are skipped, the ones executed may have a higher cost. On the other hand, the normal order (NOH) outperforms the random order (RH) in all cases except for the `Truck` and `MTG Cards` models, suggesting that domain experts followed some logic when building the feature model and defining its constraints as indicated in Section 4.2.

For large-size feature models where the process does not finish in 24 h (Table 5), the FTSH heuristic outperforms the others. FTSH explores more space than the others by analyzing fewer subtrees, although it does not find any valid subtree in most of the large models because the search space is colossal in these models. Note that when the process does not finish, the number of valid subtrees found is irrelevant for comparing the heuristics (see `Financial Services` in Table 5) because the whole search space needs to be explored/skipped and the

[7] Fig. 7 does not include the original feature model because a complex constraint may represent several (or even all) simple constraints (i.e., using the ∧ logic operator).

Table 3
Comparison of sizes (features) *w.r.t.* $\mathcal{FM}$, and execution times (seconds) for AAFM (#products and core features) over different versions of feature models.

Feature model	$\mathcal{FM}$ Size $\mathcal{F}$	$\mathcal{FM}$ Time[1] *sec*	$\mathcal{FM}_{relaxed}$ Size $\mathcal{F}$	$\mathcal{FM}_{relaxed}$ Size %	$\mathcal{FM}_{relaxed}$ Time[1] *sec*	$\mathcal{FM}_{Sans}$ Size $\mathcal{F}_{\mu}$	$\mathcal{FM}_{Sans}$ Size %	$\mathcal{FM}_{Sans}$ Time[2] *sec*	$\mathcal{GFT}$ Size $\mathcal{F}$	$\mathcal{GFT}$ Size %	$\mathcal{GFT}$ Time[2] *sec*
Tank War	37	0.03	37	0%	0.02	37	0%	1e-4	37	0%	2e-4
Mobile Media	43	0.02	43	0%	0.02	38	−12%	0.08	303	605%	2e-3
Pizzas	12	0.02	16	33%	0.02	8	−33%	0.07	131	992%	2e-3
GPL excerpt	12	0.02	16	33%	0.02	8	−33%	0.08	117	875%	8e-4
WeaFQAs	179	0.04	179	0%	0.04	155	−13%	0.20	19,713	1e4%	0.08
Truck	33	0.02	33	0%	0.02	18	−45%	0.09	745	2,158%	0.01
Apo-Games	63	0.02	72	14%	0.02	47	−25%	0.08	328	421%	2e-3
MTG Cards	781	6.74	812	4%	5.37	494	−37%	411.60	8e7	1e7%	memoryout*
JHipster	45	0.02	94	109%	0.08	26	−42%	0.16	2,027	4e3%	0.01
BerkeleyDB	76	0.02	76	0%	0.02	44	−42%	0.40	28,897	4e4%	0.18
Graphs VIZ	86	0.02	162	88%	0.34	17	−80%	0.35	5,134	5,870%	0.03
BerkeleyDB C	19	0.01	107	463%	0.05	17	−11%	0.11	522	3e3%	3e-3
GPL	66	0.04	156	136%	1.50	29	−56%	90.15	3e6	4e6%	15.86
DNN arch.	1,761	1.63	1,761	0%	1.67	832	−53%	62.08	8e6	4e5%	21.81
aXTLS	196	0.02	316	61%	0.80	72	−63%	1.97	30,925	2e4%	0.15

[1] Analysis using a *BDD*. [2] Analysis using our algorithms. * Memory blow up.

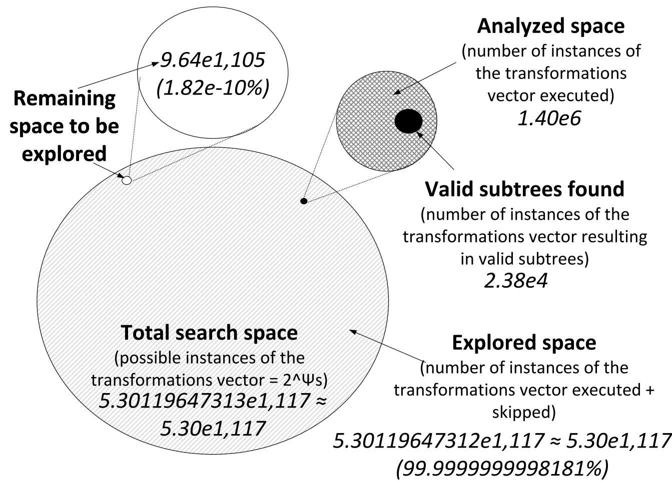
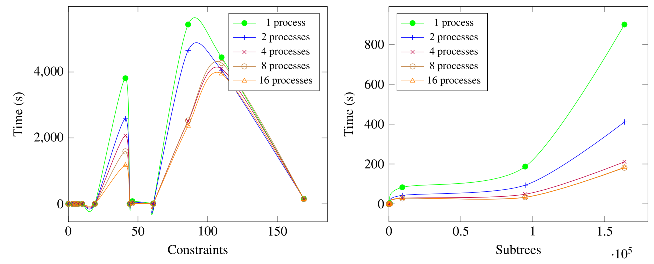


Fig. 8. Magnitude of the search space for the `uClinux-base` feature model when identifying valid subtrees.



(a) Identifying subtrees (step 3). (b) Analyzing subtrees (steps 4–6).

Fig. 9. Parallelism comparison in our approach.

final number of valid subtrees is always the same regardless the order of the constraints. In general, we conclude that the heuristics improve performance in the search for valid subtrees, reducing the time of identifying the valid subtrees and building the FM_{Sans} model. However, it is essential to be very aware of the magnitude of the numbers involved in this work (see Fig. 8). Fig. 8 illustrates the magnitude of these numbers for the `uClinux-base` model, where we can observe that despite exploring 99.99% of the total search space (5.30e1,117), the remaining space to be explored is still colossal (9.64e1,105). Note that 10^{82} is the number of atoms in the universe, and the `uClinux-base` model would have approximately $2^{3,713} \approx 5.30 * 10^{1,117}$ possible instances (subtrees) of the transformation vector.

RQ3. What are the limits for a solver based on the tree hierarchy structure of the feature tree? The scalability of our approach first depends on the number of simple constraints. When dealing with large models containing more than 200 simple constraints (see `limit 1` in Fig. 7), it becomes challenging to apply an approach that involves refactoring constraints into the feature tree. This is because of the significant increase in the number of disjoint subtrees, and in our case, in the number of possible binary vectors to be executed to identify the valid subtrees in the FM_{Sans} (step 3 of our approach). The number of valid subtrees can be huge even for medium-size models (40–200 constraints) such as MTG Cards, Financial Services, GPL, or DNN architectures models. However, the number of constraints in isolation does not appear to be a good indicator to determine the number of disjoint subtrees in the FM_{Sans}. Although removing each constraint is expected to duplicate the number of subtrees, most of the resulting subtrees are NIL, as explained in Section 4. This suggests that the internal relations of the feature trees may provide more insight into the resulting subtrees. Two cases are identified: (1) when constraints do not restrict a large configuration space too much (e.g., MTG Card or Financial Services models); and (2) when constraints restrict the configuration space too much, and thus, in the worst case, each valid subtree represents only one valid product. For instance, GPL has 107,094 products, and the FM_{Sans} has 94,770 subtrees (Table 2).

Regarding the analysis of the subtrees and the GFT, Table 3 compares the sizes and the time required to analyze small and medium-size models (regarding the number of features). For FM and $FM_{relaxed}$ we use a BDD solver, while for GFT and FM_{Sans} we use the operations over the feature trees defined in Section 6. The execution times (in seconds) include the `ProductsNumber` (Algorithm 5) and `CoreFeatures` (Algorithm 7) operations. We omit the `Valid` operation because all models are valid and the time is constant; and the `DeadFeatures` operation because no model has dead features, except for *axTLS*, and it does not make sense for an independent GFT, which never has dead features. An exhaustive performance study of each analysis operation that compares our solver with different SAT and BDD solvers is out of the scope of this paper and part of our future work. In Table 3, we can observe (highlighted in gray) that the analysis of the GFT outperforms the BDD analysis for those models with up to 5000 features (e.g., see `limit 2` in Fig. 7). From this limit, the efficiency of analyzing the GFT decreases (e.g., compare JHipster and Graphs VIZ models in Table 3). A second limit for GFT is identified at 10 million features (`limit 3` in Fig. 7), where the GFT cannot be built in memory due to its size (e.g., MTG Cards model). On the contrary, FM_{Sans} surpasses this limitation in memory since the dimensions of the subtrees (mean number of features) are smaller for all models (size column highlighted in Table 3). However, this comes at the expense of a longer analysis time because of the large number of subtrees.

Table 4
Comparison of different ordering for the constraints codification in the transformations vector to identify the valid subtrees of the FM_{Sans} (step 2 and 3). For small and medium-size models, we show the median and maximum (of 30 executions) of the analyzed subtrees and execution times, and the ratio between the number of skipped and the analyzed subtrees.

	Subtrees		Normal order (NOH)					Feature tree size (FTSH)					Random (RH)					Feature shared in constraints (FSCH)				
	Total	Valid	Analyzed subtrees			Time (s)		Analyzed subtrees			Time (s)		Analyzed subtrees			Time (s)		Analyzed subtrees			Time (s)	
Feature model	(2^{ψ_s})	$(\mathcal{ST})$	η	max	Ratio	η	max	η	max	Ratio	η	max	η	max	Ratio	η	max	η	max	Ratio	η	max
Tank War*	1	1	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A
Mobile Media	8	8	8	8	0.0	3e-3	3e-3	8	8	0.0	3e-3	4e-3	8	8	0.0	3e-3	3e-3	8	8	0.0	3-e3	5e-3
Pizzas	32	16	26	26	0.2	4e-3	7e-3	24	24	0.3	4e-3	5e-3	26	32	0.2	4e-3	6e-3	25	32	0.3	5e-3	6e-3
GPL excerpt	32	14	24	24	0.3	4e-3	4e-3	20	20	0.6	4e-3	6e-3	24	32	0.3	4e-3	6e-3	24	32	0.3	4e-3	5e-3
WeaFQAs	128	128	128	128	0.0	0.1	0.1	128	128	0.0	0.1	0.1	128	128	0.0	0.1	0.1	128	128	0.0	0.1	0.1
Truck	1024	42	182	182	4.6	5e-2	6e-2	192	192	4.3	5e-2	5e-2	167	232	5.1	4e-2	5e-2	117	148	7.8	3e-2	3e-2
Apo-Games	5e5	7	100	100	5,242	5e-2	6e-2	84	84	6,240	3e-2	4e-2	148	262	3,541	4e-2	7e-2	89	148	5,890	2e-2	4e-2
MTG Cards	2e12	163,728	1e6	1e6	2e6	1,262	1,296	1e6	1e6	2e6	2,401	2,440	1e6	3e6	2e6	1,113	2,096	8e5	2e6	3e6	1,792	4,916
JHipster	2e13	79	3,509	3,509	5e9	1.0	1.2	3,033	3,033	6e9	1.6	1.8	6,385	12,640	3e9	1.5	4.4	5,188	8,661	3e9	1.9	3.7
BerkeleyDB	7e13	310	1e5	1e5	6e8	13.6	14.1	7e4	7e4	1e9	11.7	12.0	3e4	7e4	3e9	4.5	18.8	1e4	3e4	7e9	5.2	15.7
Graphs VIZ	2e18	656	12,438	12,748	2e14	3.4	3.5	3,660	3,660	6e14	2.8	2.9	17,660	27,820	1e14	6.0	11.0	15,932	20,942	1e14	13.0	16.0
BerkeleyDB C	2e18	30	3,110	3,110	7e14	2.5	2.6	2,777	2,777	8e14	3.7	3.8	7,487	10,087	3e14	2.4	6.4	4,647	7,453	5e14	4.3	5.9
GPL	8e25	94,770	3e6	3e6	3e19	2,449	2,486	1e6	1e6	6e19	1,491	1,507	6e6	2e7	1e19	3,340	10,795	3e6	1e7	2e19	4,119	17,573
DNN arch.	1e33	9,330	6e5	6e5	2e27	6,173	8,059	3e5	3e5	4e27	3,140	3,588	2e6	5e6	7e26	9,764	15,533	1e6	3e6	1e27	12,554	52,621
aXTLS	8e50	432	3e4	3e4	3e46	209	220	1e5	1e5	5e45	1,279	1,421	1e6	2e6	6e44	3,767	4,512	1e6	9e6	6e44	9,231	33,345

* N/A: Not applicable (because this feature model has no cross-tree constraints).

Table 5
Comparison of different ordering for the constraints codification in the transformations vector to identify the valid subtrees of the FM_{Sans} (step 2 and 3). For large-size models we used a timeout of 24 h and we show the valid subtrees found from the analyzed subtrees (Analy.), the percentage of the total search space explored (Expl.), the remaining search space to be explored (To expl.), and the ratio between the number of skipped and the analyzed subtrees.

	Subtrees	Normal order (NOH)					Feature tree size (FTSH)					Random (RH)					Feature shared in constraints (FSCH)				
	Total+	Subtrees					Subtrees					Subtrees					Subtrees				
Feature model	(2^{Ψ_s})	Valid	Analy.	Expl.	To expl.	Ratio	Valid	Analy.	Expl.*	To expl.	Ratio	Valid	Analy.	Expl.	To expl.	Ratio	Valid	Analy.	Expl.	To expl.	Ratio
Financial Ser.	8e22	5e6	3e7	54%	4e22	2e15	0	2e7	92%	6e21	4e15	0	5e7	25%	6e22	4e14	0	5e7	38%	5e22	6e14
Busy Box	9e279	0	2e6	2e-19%	9e279	6e252	3e4	2e6	99%	9e279	5e251	0	3e6	1e-1%	9e279	2e270	0	2e6	51%	5e279	2e273
uClinux dist.	2e314	0	9e5	0%*	2e314	7e275	2e3	8e5	75%	5e313	2e308	1e3	2e6	4%	2e314	4e306	1e3	1e6	75%	5e313	1e308
Automotive 2.4	5e720	0	5e5	2e-10%	5e720	2e703	0	3e5	99%	3e678	2e715	0	1e6	1e-2%	5e720	6e710	0	4e5	63%	2e720	7e714
uClinux-base	5e1,117	1e4	4e6	25%	4e1,117	3e1,110	2e4	1e6	99%	1e1,106	4e1,111	1e3	3e6	32%	4e1,117	5e1,110	2e3	2e6	69%	2e1,117	2e1,111
CDL ea2468	3e2,346	0	8e5	2e-28%	3e2,346	8e2,310	0	6e4	99%	5e2,130	6e2,341	0	5e5	12%	3e2,346	9e2,339	0	3e5	94%	2e2,345	1e2,341
EMB ToolKit	3e4,630	0	4e5	69%	9e4,629	5e4,624	0	1e5	99%	5e4,587	3e4,625	0	2e5	1e.2%	3e4,630	1e4,621	0	2e5	74%	7e4,629	1e4,625
Linux 2.6.33.3	4e8,782	0	5e4	51%	2e8,782	4e8,777	†	†	†	†	†	0	6e4	5e-2%	2e8,782	2e8,774	0	7e4	69%	1e8,782	3e8,777

+ Note that the number of valid subtrees w.r.t. the total number of possible subtrees is unknown in all large-size models.
* Be careful understanding the percentages because the explored and remaining space can still be colossal (see Fig. 8 for help in understanding these huge numbers).
† The application of the heuristics does not finish due to the model's size.

Table 6
Statistical comparison of heuristics *w.r.t.* the normal order (NOH) of the constraints. We show the percentage of improvement (%) and the *p*-values for the total number of subtrees analyzed and for the execution times. A *p*-value < 0.05 means that the comparison is statistically significant at the 95% confidence level.

	Feature tree size (FTSH)				Random (RH)				Feature shared in constraints (FSCH)			
	Subtrees		Time (s)		Subtrees		Time (s)		Subtrees		Time (s)	
Feature model	%*	*p*-value	%	*p*-value	%	*p*-value	%	*p*-value	%	*p*-value	%	*p*-value
Tank War+	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A
Mobile Media	0.00	1.00	−1.52	0.33	0.00	1.00	0.12	0.46	0.00	1.00	−8.86	1.00
Pizzas	7.69	3e-15	6.94	1e-7	−1.49	4e-2	0.64	3e-2	−1.49	2e-2	−8.36	1.00
WeaFQAs	0.00	1.00	−1.22	0.97	0.00	1.00	−1.03	0.92	0.00	1.00	−4.11	1.00
Truck	−5.49	1.00	−2.12	1.00	4.06	0.12	5.00	8e-2	33.36	2e-13	40.99	7e-12
Apo-Games	16.00	3e-15	28.66	8e-10	−53.68	1.00	−13.27	0.76	7.52	1e-5	46.44	4e-11
MTG Cards	16.90	3e-15	−89.88	1.00	9.22	1e-3	9.98	1e-2	35.94	3e-9	−56.18	0.99
JHipster	13.57	3e-15	−58.74	1.00	−93.43	1.00	−74.86	1.00	−50.38	1.00	−98.28	1.00
BerkeleyDB	42.93	3e-15	13.48	7e-12	73.13	3e-13	62.28	2e-10	89.69	3e-13	56.49	1e-10
Graphs VIZ	70.60	5e-15	16.81	7e-12	−46.76	1.00	−104.36	1.00	−27.63	1.00	−280.92	1.00
BerkeleyDB C	10.71	3e-15	−47.39	1.00	−131.81	1.00	−5.02	6e-2	−52.85	1.00	−70.57	1.00
GPL	54.79	3e-15	39.17	7e-12	−131.38	1.00	−81.37	0.99	−48.22	0.76	−133.81	1.00

* Positive values represent an improvement *w.r.t.* normal order (NOH), and negative values a deterioration.
+ N/A: Not applicable (because this feature model has no cross-tree constraints).

RQ4. To what extent do parallelization techniques improve the automated analysis of feature trees? Fig. 9 shows how parallelization improves the performance of (a) the identification of the valid subtrees in FM_{Sans}(step 3), and (b) the analysis of those subtrees (steps 4–6). We can observe that for identifying the subtrees, the performance depends not only on the number of constraints that determines the number of transformations (length of binary vectors), but also on the number of features and their relations that determine the number of transformations that result in NIL. For instance, see the increment in time for the MTG Cards model with only 41 constraints but with 812 features (i.e., an unrestricted feature tree results in thousands of subtrees); in contrast to the axTLS model with 169 constraints and 316 features (i.e., a constrained feature tree results in fewer subtrees because increment the number of binary vectors resulting in NIL). Multiprocessing drastically reduces the time of identifying the subtrees (e.g., from 90 min using only one process to 42 min using 16 processes in the GPL model). For large-scale models (> 200 constraints) and some medium-size models (e.g., Financial Services) the number of vectors is impracticable, even for a supercomputer. We can observe in Table 2 the percentage of binary vectors explored in 24 h. However, such percentages are not representative of the time required to complete the FM_{Sans} because of the colossal numbers of remaining vectors and the unknown number of skipped vectors caused by the empty subtrees found.

In the analysis of the subtrees, the performance depends on the number of constraints (length of the binary transformations vector) and on the size of the feature tree because we know that the vector will be wholly executed, resulting in a valid subtree. Parallelism, in this case, is based on spawning a fixed pool of processes executing the same task (the transformations vector) with different parameters (binary value), and there is a trade-off between the benefits of a high number of processes and the cost of creating those processes. We can observe in Fig. 9(b) that the benefits of increasing the number of processes reduce from 8 processes (i.e., approximately the number of cores in the computer).

7.4. Threats to validity

We discuss the different threats to validity that may affect our evaluation (Wohlin et al., 2012).

External validity.

- *Comparison with traditional solvers in AAFMs.* While we demonstrate that some analysis operations outperform BDDs for small models, a full comparison of all analysis operations using FM_{Sans} against traditional solvers like SAT, #SAT (Sundermann et al., 2023), or BDDs (Heradio et al., 2019) is beyond the scope of this paper. The lack of a detailed comparison stems from several reasons: (1) not all analysis operations can be efficiently implemented with each solver type, and in some cases, it does not make sense to do so (e.g., SAT solvers are inefficient for counting-based operations like *ProductsNumber* requiring to enumerate all solutions), and (2) for a fair comparison, we would need an implementation of the analysis operations of #SAT solvers in Python, the language used in our approach. Unfortunately, there are none, including Flamapy (Galindo et al., 2023). However, we open the possibility for future research to explore how transformations of feature models and the use of solvers could leverage parallel computation, which may lead to significant improvements in the AAFM field.

Internal validity.

- *Experimentation set-up.* A threat to internal validity is our choice of feature models and their sizes and characteristics for the evaluation corpus, as well as the configuration parameters for the experiments. The chosen models were used in other related work analyzing and removing constraints (Knüppel et al., 2017; Knüppel, 2016) and have been complemented by models often used to evaluate SPLs (Lopez-Herrejon and Batory, 2001; Horcas et al., 2023d). Regarding parameters for the heuristic FSCH, we used the default values of the genetic algorithm, but a deep analysis of this heuristic with different parameters is out of scope of this paper and it is expected that results dot no affect conclusions raised in our results.

Construct validity.

- *Feature tree structure and constraints in feature models.* The structure of the feature tree and the relationship between the different constraints in each feature model may affect the analysis of some heuristics and affect the performance of the functions applied (e.g., `Commitment` and `Deletion` functions). To mitigate this, we have selected a wide set of feature models varying in size and complexity.
- *Heuristics overhead.* The time to apply the heuristic has not been taken into account since it depends on the nature of each heuristic, and for the proposed heuristics and small and medium size models it can be considered negligible. However, for large-scale models, the application of some heuristics (e.g., the equations of heuristic FTSH) can be challenging, as occurs for the `Linux 2.6.33.3` model with 29,175 constraints (Table 5). The normal order heuristic (NOH) requires no additional time. The application of heuristics FTSH, RH, and FSCH is linear as they only require reordering the vector based on the equations of FTSH, randomly in the case of RH, or based on the constraint analysis of FSCH. The reordering of the vector takes place before applying step 2 of the process of identifying valid subtrees. Regarding the heuristic FSCH, although the constraint analysis may require

more time due to the genetic algorithm, it is run offline only once regardless of the FM, storing the result of the constraint analysis similarly to Table 1 and reusing the information for any feature model. In this case, an analysis has been carried out for combinations of three constraints ($m = 3$).

Conclusion validity.

- *Reliability and robustness of measures.* We address conclusion validity by executing 30 independent runs of each experiment and applying standard statistical analysis techniques (e.g., Mann–Whitney U test). For large models where the experiments are not viable to be executed multiple times, we set a timeout of 24 h and check the status of the process at that point. The goal, in this case, is to have an idea about the scalability limits of our approach instead of a fair comparison of models, heuristics, and/or parallelization.
- *Transformation of constraints to CNF.* A potential limitation of our approach lies in the requirement to transform complex constraints into CNF as a prerequisite for applying Knüppel's algorithm in Step 1. Although CNF is necessary for the distinction between pseudo-complex and strict-complex constraints required by the Knüppel's algorithm, the process of converting general-form constraints to CNF can be computationally challenging (Kuiter et al., 2022). Specifically, the transformation can lead to an exponential increase in the number of terms, which may negatively impact the efficiency and scalability of the approach. Although techniques such as the Tseytin transformation (Kuiter et al., 2022) can mitigate some of these issues by introducing auxiliary variables to control the growth of terms, the complexity of dealing with these auxiliary variables in our approach remains a open challenge.
- *Constraint removal in large-scale models.* For huge feature models with a large number of constraints (such as the Automotive model Knüppel et al., 2017 with 2394 constraints or the Linux model Knüppel et al., 2017 with 29,175 constraints), trying to remove the constraints may be counterproductive. In our approach, the complexity shifts from constraints to the structure itself, similar to what BDDs do. While our approach shows improvements in analysis times, the benefits may diminish for huge models, and the scalability of our method requires further evaluation, especially when compared to optimized BDD techniques. The real potential of our approach lies in the possibility of parallelizing the analysis operations using the transformed model (FM_{Sans}), which could open new windows for research in AAFM.

8. Related work

SPL community has put little emphasis on the elimination of constraints from feature models, mainly because of two reasons: (1) Constraints play a crucial role in compacting feature models (Knüppel et al., 2017), and (2) the high cost of eliminating the constraints (Gil et al., 2010). Five main works focus on eliminating constraints (van den Broek et al., 2008; van den Broek and Galvão, 2009; Gil et al., 2010; Knüppel et al., 2017; Knüppel, 2016). Chronologically, Broek et al. focused on eliminating simple constraints (van den Broek et al., 2008) and then on analyzing the resulting generalized feature tree (van den Broek and Galvão, 2009). Gil et al. (2010) addressed the problem of removing constraints from a theoretical perspective by following the approach of Czarnecki and Wasowski (2007) of translating feature models to logic and back, and proving that all constraints can be eliminated for the price of introducing a new set of features. The work of Knüppel et al. (2017) and his Master's Thesis (Knüppel, 2016) focus on the elimination of complex constraints using abstract features and simple constraints. Table 7 compares in detail these works and our approach.

Regarding the AAFMs using the hierarchical structure of the feature tree, in contrast to the use of SAT solving (Liang et al., 2015), #SAT (Sundermann et al., 2023), CSP (Benavides et al., 2010) or BDD (Heradio et al., 2016), also van den Broek and Galvão (2009) proposed some analysis operations as our Algorithms 4–8, but implemented in the obsolete functional language Miranda (Turner, 1985), which makes it difficult to apply their algorithms in current feature models technologies (Horcas et al., 2023b), such as FeatureIDE (Thüm et al., 2014b), pure::variants (Beuche, 2019), or Flama (Galindo et al., 2023). There are other hybrid approaches (Mendonça, 2009; Fernández-Amorós et al., 2014) that rely on a reasoning engine to solve the textual constraints, while another module works on the feature tree. For instance, Fernández-Amorós et al. (2014) propose *treecount*, an algorithm for model counting that traverses the feature tree while considering textual constraints using a reasoning engine to avoid repeating the exact computations. Similarly, Mendonça (2009) present *Feature Model Reasoning System* (FMRS) which performs constraints propagation and backtracking inside the tree. These approaches (Mendonça, 2009; Fernández-Amorós et al., 2014) only support specific operations, such as model counting or commonality calculations (Fernández-Amorós et al., 2014).

Parallelization techniques have also been used for the AAFMs. Galindo et al. (2016) considered the problem of exploiting the result of enumerating all feature model configurations using distributed computing techniques. Despite an enumeration-based approach being simply impossible for large configuration spaces, using distributed and parallel techniques look promising as we present in this paper without the need to enumerate the products. For instance, the number of disjoint subtrees in our approach is much smaller in general than the number of products in the feature model (Table 2). Others authors use parallelization techniques for conflict detection and diagnosis (Vidal et al., 2021), prioritizing software testing (Lopez-Herrejon et al., 2014), multi-objective combinatorial optimization (Guo et al., 2014), and evolutionary configuration (Shi et al., 2018, 2019).

Finally, scalability is one of the main issues of automatic analysis proposals regarding either time or space restrictions. For that reason, there is a considerable number of studies about the scalability of solvers (Sundermann et al., 2020; Sprey et al., 2020), of BDDs (Heß et al., 2021), and of the proposals eliminating constraints (van den Broek et al., 2008; van den Broek and Galvão, 2009; Gil et al., 2010; Knüppel et al., 2017; Knüppel, 2016). In the case of solvers, in (Sundermann et al., 2020) authors analyze the scalability of 9 #SAT solvers on 127 industrial FMs available in the FeatureIDE tool for the count operation and the main conclusions are that there are large models for which none of the evaluated solvers scale well; there is not always a correlation of size and scalability; some large systems scale better than other smaller ones, and no one solver performs better than the other solvers in every model. In (Sprey et al., 2020) authors compare the scalability of two SMT solvers and one SAT solver on 117 real-world FMs. The main conclusions are that SMT is more expressive than SAT and can be used to solve more complex problems; each solver is faster or not depending on the type of operation; SMT is faster in generating explanations, while the SAT solver outperformed SMT solvers in other analysis such as satisfiability requests and automated decision propagation, and the combination of SAT and SMT solvers can be useful to improve performance.

For BBDs the main scalability problem is related to the size of the BDD constructed to represent the FM and therefore the memory requested to store it. In (Heß et al., 2021), authors present a performance evaluation of state-of-the-art BDD libraries. The main conclusion of this work is that BDDs have a serious scalability problem for large and complex FMs since neither of the BDD libraries scales well. Thus, although BDDs are a good alternative to SAT and #SAT solvers, their performance needs to be improved. Similarly to our work, the approach followed by BDD solvers to improve scalability is the use of heuristics for compiling FMs to BDDs reducing the size of the BDDs produced (Mendonça et al., 2008). In the general case, these heuristics define different variable ordering that can help to reduce

Table 7
Comparison of the main related work addressing the elimination of constraints in feature models (FMs).

van den Broek et al. (2008), van den Broek and Galvão (2009)	Gil et al. (2010)	Knüppel et al. (2017), Knüppel (2016)	Our approach (2024)
Contributions. Concept of generalized feature tree (GFT). • Algorithms to eliminate simple constraints in FMs. • Algorithms for computing analysis operations on the hierarchical structure of the GFT.	**Contributions.** Removal of constraints by transforming FMs to logic and then to feature trees. • It shows that the elimination of constraints provides an efficient solution for the feature editing problem.	**Contributions.** Formalization of relaxed FMs. • Product-preserving algorithm to eliminate complex constraints. • Demonstration that FMs with complex constraints and relaxed FMs are equally expressive.	**Contributions.** Formalization of the FM_{Sans}. • Elimination of both (complex and simple) constraints. • Solution to the scalability problem in size. • Algorithms for parallel automated analysis on FM_{Sans}.
Problem addressed. Need for an efficient and effective way to compute properties of SPLs based on the hierarchical structure of FMs instead of using solvers.	**Problem addressed.** Inter-dependencies between features in FMs which are not captured by the feature tree diagram. • Feature editing problem.	**Problem addressed.** Whether less expressive FMs, which only simple constraints, are sufficient to manage real-world product lines.	**Problem addressed.** Whether all constraints can be removed without new auxiliary features. • Scalability in size of the removal of constraints in FMs.
Inputs. A FM with only simple constraints.	**Inputs.** Arbitrary FMs.	**Inputs.** Arbitrary FMs with complex constraints.	**Inputs.** Arbitrary FMs.
Outputs. A generalized feature tree (GFT).	**Outputs.** A generalized feature tree (GFT).	**Outputs.** Relaxed FMs with simple constraints.	**Outputs.** FM_{Sans} model and GFT.
Limitations. Do not consider complex constraints. • Exponential increase in the size of the GFT with duplicated features. • No practical evaluation.	**Limitations.** Inevitable exponential increase in the size of the tree. • Theoretical work where practical limitations are not explicitly discussed.	**Limitations.** A significant increment in the number of auxiliary abstract features and simple constraints. • It does not address the issue of scalability in size.	**Limitations.** Performance detriment in the analysis of large-scale FMs. • Need of high computing resources to make it effective for large-scale FMs.

the tree size. In the particular case of using BDDs to represent FMs they also consider constraint ordering (Popov et al., 2023). In Popov et al. (2023) authors overview existing approaches for minimizing BDD size and propose some modifications of existing approaches to improve them by considering constraint ordering. Beyond BDDs, Müller et al. (2000) also deal with constraints ordering over feature trees and provide a new order relation based on the information carried by the constraints. However, this new relation is very dependent on the type of the constraints (i.e., simple, pseudo-complex, strict-complex).

For the works that focus on removing constraints (van den Broek et al., 2008; van den Broek and Galvão, 2009; Gil et al., 2010; Knüppel et al., 2017; Knüppel, 2016), they suffer from a significant increase in the size of the resulting tree, in terms of both the number of features and the number of simple constraints (in approaches where complex constraints are transformed into simple ones). For instance, in Knüppel et al. (2017), authors analyze more than 100 variability models and they report an increase in the number of features of 58% on average and an increase in the number of simple constraints of 74% on average. In the worst case, for the Linux Kernel, the increase is 582% and 713% respectively. Also, in van den Broek et al. (2008), van den Broek and Galvão (2009), Gil et al. (2010) authors claim how the computational complexity of algorithms increases exponentially for some variability models.

9. Conclusions and future work

We have presented an approach for completely removing all constraints in feature models without considering a new set of features. Our knowledge compilation technique (FM_{Sans}) alleviates the exponential blow-up in memory in the elimination of constraints and allows analyzing those models whose GFT cannot be built due to their size. With this contribution, we show the practical applicability of the idea of splitting the semantics of the feature model into disjoint subtrees to be analyzed in parallel. We have also shown that reordering the constraints in the feature model may improve the performance of identifying the disjoint subtrees in the construction of the FM_{Sans}. We envision that other AAFM techniques can benefit from our approach to dividing the complexity of large feature models and improving their analyses through reorganization of the constraints and parallelization.

Software artifact and open science

Artifact repository: https://github.com/CAOSD-group/fmsans

CRediT authorship contribution statement

Jose-Miguel Horcas: Writing – original draft, Visualization, Software, Methodology, Investigation, Formal analysis, Conceptualization. **Joaquín Ballesteros:** Validation, Software, Resources, Methodology, Data curation. **Mónica Pinto:** Writing – review & editing, Visualization, Supervision, Investigation. **Lidia Fuentes:** Writing – review & editing, Supervision, Resources, Project administration, Methodology, Funding acquisition.

Declaration of competing interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

Acknowledgments

Work supported by the projects *TASOVA PLUS* research network (RED2022-134337-T), and *IRIS* (PID2021-122812OB-I00) (co-financed by FEDER funds, Spain); and by Universidad de Málaga, Spain.

Data availability

I have shared the link to my data in the paper.

References

Åkesson, J., Nilsson, S., Krüger, J., Berger, T., 2019. Migrating the android apo-games into an annotation-based software product line. In: 23rd International Systems and Software Product Line Conference, Vol. A. SPLC, ACM, pp. 19:1–19:5. http://dx.doi.org/10.1145/3336294.3342362.

Apel, S., Batory, D.S., Kästner, C., Saake, G., 2013. Feature-Oriented Software Product Lines - Concepts and Implementation. Springer, http://dx.doi.org/10.1007/978-3-642-37521-7.

Arcuri, A., Briand, L.C., 2014. A Hitchhiker's guide to statistical tests for assessing randomized algorithms in software engineering. Softw. Test. Verif. Reliab. 24 (3), 219–250.

Audemard, G., Simon, L., 2018. On the glucose SAT solver. Int. J. Artif. Intell. Tools 27 (1), http://dx.doi.org/10.1142/S0218213018400018.

Batory, D.S., 2005. Feature models, grammars, and propositional formulas. In: 9th International Conference on Software Product Lines. SPLC, In: LNCS, vol. 3714, Springer, pp. 7–20. http://dx.doi.org/10.1007/11554844_3.

Benavides, D., Segura, S., Cortés, A.R., 2010. Automated analysis of feature models 20 years later: A literature review. Inf. Syst. 35 (6), 615–636. http://dx.doi.org/10.1016/j.is.2010.01.001.

Benavides, D., Sundermann, C., Feichtinger, K., Galindo, J.A., Rabiser, R., Thüm, T., 2025. UVL: feature modelling with the universal variability language. Journal of Systems and Software (ISSN: 0164-1212) 225, 112326. http://dx.doi.org/10.1016/j.jss.2024.112326.

Berger, T., Pfeiffer, R., Tartler, R., Dienst, S., Czarnecki, K., Wasowski, A., She, S., 2014. Variability mechanisms in software ecosystems. Inf. Softw. Technol. 56 (11), 1520–1535. http://dx.doi.org/10.1016/j.infsof.2014.05.005.

Beuche, D., 2019. Industrial variant management with pure: : variants. In: 23rd International Systems and Software Product Line Conference, Vol. B. SPLC, ACM, pp. 64:1–64:3. http://dx.doi.org/10.1145/3307630.3342391.

Camelo, M., Gramaglia, M., Soto, P., Fuentes, L., Ballesteros, J., Nogueras, A.B., Garcia-Aviles, G., Latré, S., Garcia-Saavedra, A., Fiore, M., 2022. DAEMON: a network intelligence plane for 6G networks. In: IEEE Globecom 2022 Workshops. IEEE, pp. 1341–1346. http://dx.doi.org/10.1109/GCWkshps56602.2022.10008662.

Czarnecki, K., Helsen, S., Eisenecker, U.W., 2005. Formalizing cardinality-based feature models and their specialization. Softw. Process. Improv. Pr. 10 (1), 7–29. http://dx.doi.org/10.1002/spip.213.

Czarnecki, K., Wasowski, A., 2007. Feature diagrams and logics: There and back again. In: 11th International Software Product Lines Conference. SPLC, IEEE, Kyoto, Japan, pp. 23–34. http://dx.doi.org/10.1109/SPLINE.2007.24.

Darwiche, A., Marquis, P., 2002. A knowledge compilation map. J. Artificial Intelligence Res. 17, 229–264.

Fernández-Amorós, D., Heradio, R., Cerrada, J.A., Cerrada, C., 2014. A scalable approach to exact model and commonality counting for extended feature models. IEEE Trans. Softw. Eng. 40 (9), 895–910. http://dx.doi.org/10.1109/TSE.2014.2331073.

Fortin, F.-A., De Rainville, F.-M., Gardner, M.-A., Parizeau, M., Gagné, C., 2012. DEAP: Evolutionary algorithms made easy. J. Mach. Learn. Res. 13, 2171–2175.

Galindo, J.A., Acher, M., Tirado, J.M., Vidal, C., Baudry, B., Benavides, D., 2016. Exploiting the enumeration of all feature model configurations: a new perspective with distributed computing. In: 20th International Systems and Software Product Line Conference. SPLC, ACM, Beijing, China, pp. 74–78. http://dx.doi.org/10.1145/2934466.2934478.

Galindo, J.A., Horcas, J.M., Felfernig, A., Fernández-Amorós, D., Benavides, D., 2023. FLAMA: a collaborative effort to build a new framework for the automated analysis of feature models. In: 27th ACM International Systems and Software Product Line Conference, Vol. B. SPLC, ACM, pp. 16–19. http://dx.doi.org/10.1145/3579028.3609008.

Ghamizi, S., Cordy, M., Papadakis, M., Traon, Y.L., 2019. Automated search for configurations of convolutional neural network architectures. In: 23rd International Systems and Software Product Line Conference, Vol. A. SPLC, ACM, pp. 21:1–21:12. http://dx.doi.org/10.1145/3336294.3336306.

Gil, Y., Kremer-Davidson, S., Maman, I., 2010. Sans constraints? Feature diagrams vs. Feature models. In: 14th International Software Product Lines Conference (SPLC): Going beyond, Vol. 6287. Springer, pp. 271–285. http://dx.doi.org/10.1007/978-3-642-15579-6_19.

Gramaglia, M., Camelo, M., Fuentes, L., Ballesteros, J., Baldoni, G., Cominardi, L., Garcia-Saavedra, A., Fiore, M., 2022. Network intelligence for virtualized RAN orchestration: The DAEMON approach. In: Joint European Conference on Networks and Communications & 6G Summit. EuCNC/6G Summit, pp. 482–487. http://dx.doi.org/10.1109/EuCNC/6GSummit54941.2022.9815816.

Guo, J., Zulkoski, E., Olaechea, R., Rayside, D., Czarnecki, K., Apel, S., Atlee, J.M., 2014. Scaling exact multi-objective combinatorial optimization by parallelization. In: 29th IEEE/ACM International Conference on Automated Software Engineering. ASE, ACM, Vasteras, Sweden, pp. 409–420. http://dx.doi.org/10.1145/2642937.2642971.

Halin, A., Nuttinck, A., Acher, M., Devroey, X., Perrouin, G., Baudry, B., 2019. Test them all, is it worth it? Assessing configuration sampling on the JHipster Web development stack. Empir. Softw. Eng. 24 (2), 674–717. http://dx.doi.org/10.1007/s10664-018-9635-4.

Hentze, M., Sundermann, C., Thüm, T., Schaefer, I., 2022. Quantifying the variability mismatch between problem and solution space. In: 25th International Conference on Model Driven Engineering Languages and Systems. MODELS, pp. 322–333. http://dx.doi.org/10.1145/3550355.3552411.

Heradio, R., Fernández-Amorós, D., Galindo, J.A., Benavides, D., Batory, D.S., 2022. Uniform and scalable sampling of highly configurable systems. Emp. Soft. Eng. 27 (2), http://dx.doi.org/10.1007/s10664-021-10102-5.

Heradio, R., Fernández-Amorós, D., Mayr-Dorn, C., Egyed, A., 2019. Supporting the statistical analysis of variability models. In: 41st International Conference on Software Engineering. ICSE, IEEE / ACM, Montreal, Canada, pp. 843–853. http://dx.doi.org/10.1109/ICSE.2019.00091.

Heradio, R., Perez-Morago, H., Fernández-Amorós, D., Bean, R., Cabrerizo, F.J., Cerrada, C., Herrera-Viedma, E., 2016. Binary decision diagram algorithms to perform hard analysis operations on variability models. In: 15th New Trends in Software Methodologies, Tools and Techniques. SoMeT, In: Frontiers in Artificial Intelligence and Applications, vol. 286, pp. 139–154. http://dx.doi.org/10.3233/978-1-61499-674-3-139.

Heß, T., Sundermann, C., Thüm, T., 2021. On the scalability of building binary decision diagrams for current feature models. In: 25th ACM International Systems and Software Product Line Conference, Vol. A. SPLC, ACM, pp. 131–135. http://dx.doi.org/10.1145/3461001.3474452.

Horcas, J.M., 2018. WeaFQAs: A Software Product Line Approach for Customizing and Weaving Efficient Functional Quality Attributes (Ph.D. thesis). Universidad de Málaga, URL https://hdl.handle.net/10630/17231.

Horcas, J.M., Ballesteros, J., Pinto, M., Fuentes, L., 2023a. Elimination of constraints for parallel analysis of feature models. In: 27th ACM International Systems and Software Product Line Conference, Vol. A. SPLC, ACM, pp. 99–110. http://dx.doi.org/10.1145/3579027.3608981.

Horcas, J.M., Galindo, J.A., Benavides, D., 2022. Variability in data visualization: a software product line approach. In: 26th ACM International Systems and Software Product Line Conference, Vol.A. SPLC, ACM, Graz, Austria, pp. 55–66. http://dx.doi.org/10.1145/3546932.3546993.

Horcas, J.M., Pinto, M., Fuentes, L., 2023b. Empirical analysis of the tool support for software product lines. Softw. Syst. Model. 22 (1), 377–414. http://dx.doi.org/10.1007/s10270-022-01011-2.

Horcas, J.M., Pinto, M., Fuentes, L., 2023c. A modular metamodel and refactoring rules to achieve software product line interoperability. J. Syst. Softw. 197, 111579. http://dx.doi.org/10.1016/j.jss.2022.111579.

Horcas, J.M., Strüber, D., Burdusel, A., Martinez, J., Zschaler, S., 2023d. We're not gonna break it! consistency-preserving operators for efficient product line configuration. IEEE Trans. Softw. Eng. 49 (3), 1102–1117. http://dx.doi.org/10.1109/TSE.2022.3171404.

Ignatiev, A., Morgado, A., Marques-Silva, J., 2018. PySAT: A python toolkit for prototyping with SAT oracles. In: 21st International Conference on Theory and Applications of Satisfiability Testing, Vol.10929. SAT, Springer, pp. 428–437. http://dx.doi.org/10.1007/978-3-319-94144-8_26.

Kang, K.C., Cohen, S.G., Hess, J.A., Novak, W.E., Peterson, A.S., 1990. Feature-Oriented Domain Analysis (FODA) Feasibility Study. Tech. Rep., Carnegie-Mellon University, CMU/SEI-90-TR-21.

Knüppel, A., 2016. The Role of Complex Constraints in Feature Modeling (Master's thesis). Technische Universität Braunschweig, URL https://www.isf.cs.tu-bs.de/cms/team/knueppel/downloads/thesisKnueppel16.pdf.

Knüppel, A., Thüm, T., Mennicke, S., Meinicke, J., Schaefer, I., 2017. Is there a mismatch between real-world feature models and product-line research? In: 11th Joint Meeting on Foundations of Software Engineering. ESEC/FSE, pp. 291–302. http://dx.doi.org/10.1145/3106237.3106252.

Kowal, M., Ananieva, S., Thüm, T., 2016. Explaining anomalies in feature models. In: 15th ACM SIGPLAN International Conference on Generative Programming (GPCE): Concepts and Experiences. ACM, pp. 132–143. http://dx.doi.org/10.1145/2993236.2993248.

Krieter, S., 2020. Large-scale T-wise interaction sampling using YASA. In: 24th ACM International Systems and Software Product Line Conference, Vol. A. SPLC, pp. 29:1–29:4. http://dx.doi.org/10.1145/3382025.3414989.

Kuiter, E., Krieter, S., Sundermann, C., Thüm, T., Saake, G., 2022. Tseitin or not tseitin? The impact of CNF transformations on feature-model analyses. In: 37th IEEE/ACM International Conference on Automated Software Engineering. ASE, ACM, pp. 110:1–110:13. http://dx.doi.org/10.1145/3551349.3556938.

Liang, J.H., Ganesh, V., Czarnecki, K., Raman, V., 2015. SAT-based analysis of large real-world feature models is easy. In: 19th International Conference on Software Product Line. SPLC, ACM, New York, NY, USA, pp. 91–100. http://dx.doi.org/10.1145/2791060.2791070.

Lopez-Herrejon, R.E., Batory, D.S., 2001. A standard problem for evaluating product-line methodologies. In: 3rd International Conference on Generative and Component-Based Software Engineering. GCSE, In: LNCS, vol. 2186, Springer, pp. 10–24. http://dx.doi.org/10.1007/3-540-44800-4_2.

Lopez-Herrejon, R.E., Ferrer, J., Chicano, F., Haslinger, E.N., Egyed, A., Alba, E., 2014. A parallel evolutionary algorithm for prioritized pairwise testing of software product lines. In: 16th Genetic and Evolutionary Computation Conference. GECCO, ACM, Vancouver, BC, Canada, pp. 1255–1262. http://dx.doi.org/10.1145/2576768.2598305.

Martinez, J., Tërnava, X., Ziadi, T., 2018. Software product line extraction from variability-rich systems: the robocode case study. In: 22nd International Systems and Software Product Line Conference, Vol. 1. SPLC, ACM, pp. 132–142. http://dx.doi.org/10.1145/3233027.3233038.

Mendonça, M., 2009. Efficient Reasoning Techniques for Large Scale Feature Models (Ph.D. thesis). University of Waterloo, URL https://hdl.handle.net/10012/4201.

Mendonça, M., Wasowski, A., Czarnecki, K., 2009. SAT-based analysis of feature models is easy. In: 13th International Software Product Lines Conference, Vol. 446. SPLC, ACM, pp. 231–240, URL https://dl.acm.org/citation.cfm?id=1753267.

Mendonça, M., Wasowski, A., Czarnecki, K., Cowan, D.D., 2008. Efficient compilation techniques for large scale feature models. In: 7th International Conference on Generative Programming and Component Engineering. GPCE, ACM, pp. 13–22. http://dx.doi.org/10.1145/1449913.1449918.

Müller, M., Niehren, J., Podelski, A., 2000. Ordering constraints over feature trees. Constraints an Int. J. 5 (1/2), 7–41. http://dx.doi.org/10.1023/A:1009866317252.

Oh, J., Gazzillo, P., Batory, D.S., 2019. *t*-wise coverage by uniform sampling. In: 23rd International Systems and Software Product Line Conference, Vol. A. SPLC, ACM, pp. 15:1–15:4. http://dx.doi.org/10.1145/3336294.3342359.

Pett, T., Krieter, S., Runge, T., Thüm, T., Lochau, M., Schaefer, I., 2021. Stability of product-line samplingin continuous integration. In: 15th International Working Conference on Variability Modelling of Software-Intensive Systems. VaMoS, pp. 18:1–18:9. http://dx.doi.org/10.1145/3442391.3442410.

Popov, M., Tomás, B., Iser, M., Ostertag, T., 2023. Construction of decision diagrams for product configuration. In: 25th International Configuration Workshop, Vol. 3509. ConfWS, pp. 108–117, URL https://ceur-ws.org/Vol-3509/paper15.pdf.

Schmitt, A., Rock, G., Bettinger, C., 2018. Glencoe – A tool for specification, visualization and formal analysis of product lines. In: 25th International Conference on Transdisciplinary Engineering, Vol. 7. Modena, Italy, pp. 665–673. http://dx.doi.org/10.3233/978-1-61499-898-3-66.

Schobbens, P., Heymans, P., Trigaux, J., Bontemps, Y., 2007. Generic semantics of feature diagrams. Comput. Networks 51 (2), 456–479. http://dx.doi.org/10.1016/j.comnet.2006.08.008.

She, S., Lotufo, R., Berger, T., Wasowski, A., Czarnecki, K., 2010. The variability model of the linux kernel. In: 4th International Workshop on Variability Modelling of Software-Intensive Systems. VaMoS, pp. 45–51, URL http://www.vamos-workshop.net/proceedings/VaMoS_2010_Proceedings.pdf.

Shi, K., Yu, H., Guo, J., Fan, G., Chen, L., Yang, X., 2019. A parallel framework of combining satisfiability modulo theory with indicator-based evolutionary algorithm for configuring large and real software product lines. Int. J. Softw. Eng. Knowl. Eng. 29 (4), 489–513. http://dx.doi.org/10.1142/S0218194019500219.

Shi, K., Yu, H., Guo, J., Fan, G., Yang, X., 2018. A parallel portfolio approach to configuration optimization for large software product lines. Softw. Pr. Exp. 48 (9), http://dx.doi.org/10.1002/spe.2594.

Siegmund, N., Rosenmüller, M., Kuhlemann, M., Kästner, C., Apel, S., Saake, G., 2012. SPL Conqueror: Toward optimization of non-functional properties in software product lines. Softw. Qual. J. 20 (3–4), 487–517. http://dx.doi.org/10.1007/s11219-011-9152-9.

Sprey, J., Sundermann, C., Krieter, S., Nieke, M., Mauro, J., Thüm, T., Schaefer, I., 2020. SMT-based variability analyses in FeatureIDE. In: VaMoS '20. ACM, New York, NY, USA, http://dx.doi.org/10.1145/3377024.3377036.

Sundermann, C., Brancaccio, V.F., Kuiter, E., Krieter, S., Heß, T., Thüm, T., 2024. Collecting feature models from the literature: A comprehensive dataset for benchmarking. In: 28th ACM International Systems and Software Product Line Conference, Vol. A. SPLC, ACM, pp. 54–65. http://dx.doi.org/10.1145/3646548.3672590.

Sundermann, C., Heß, T., Nieke, M., Bittner, P.M., Young, J.M., Thüm, T., Schaefer, I., 2023. Evaluating state-of-the-art # SAT solvers on industrial configuration spaces. Empir. Softw. Eng. 28 (2), 29. http://dx.doi.org/10.1007/s10664-022-10265-9.

Sundermann, C., Nieke, M., Bittner, P.M., Heß, T., Thüm, T., Schaefer, I., 2021b. Applications of #sat solvers on feature models. In: 15th International Working Conference on Variability Modelling of Software-Intensive Systems. VaMoS, pp. 12:1–12:10. http://dx.doi.org/10.1145/3442391.3442404.

Sundermann, C., Thüm, T., Schaefer, I., 2020. Evaluating #sat solvers on industrial feature models. In: VaMoS '20. Association for Computing Machinery, New York, NY, USA, http://dx.doi.org/10.1145/3377024.3377025.

Thüm, T., 2020. A BDD for Linux?: the knowledge compilation challenge for variability. In: 24th ACM International Systems and Software Product Line Conference, Vol. A. SPLC, pp. 16:1–16:6. http://dx.doi.org/10.1145/3382025.3414943.

Thüm, T., Apel, S., Kästner, C., Schaefer, I., Saake, G., 2014a. A classification and survey of analysis strategies for software product lines. ACM Comput. Surv. 47 (1), 6:1–6:45. http://dx.doi.org/10.1145/2580950.

Thüm, T., Kästner, C., Benduhn, F., Meinicke, J., Saake, G., Leich, T., 2014b. FeatureIDE: An extensible framework for feature-oriented software development. Sci. Comput. Program. 79, 70–85. http://dx.doi.org/10.1016/j.scico.2012.06.002.

Tieber, R., Felfernig, A., 2021. A knowledge-based configurator for building magic: The gathering card decks. In: 23rd International Configuration Workshop, Vol. 2945. ConfWS, CEUR-WS.org, pp. 55–57, URL https://ceur-ws.org/Vol-2945/42-RT-ConfWS21_paper_3.pdf.

Turner, D., 1985. Miranda: A non-strict functional language with polymorphic types. In: Functional Programming Languages and Computer Architecture (FPCA). In: LNCS, vol. 201, Springer, pp. 1–16. http://dx.doi.org/10.1007/3-540-15975-4_26.

van den Broek, P., Galvão, I., 2009. Analysis of feature models using generalised feature trees. In: 3rd International Workshop on Variability Modelling of Software-Intensive Systems, Vol .29. VaMoS, pp. 29–35, URL http://www.vamos-workshop.net/proceedings/VaMoS_2009_Proceedings.pdf.

van den Broek, P., Galvão, I., Noppen, J., 2008. Elimination of constraints from feature trees. In: Workshop on Analyses of Software Product Lines (ASPL) @ SPLC'08 Second (Workshops). pp. 227–232.

Vidal, C., Felfernig, A., Galindo, J.A., Atas, M., Benavides, D., 2021. Explanations for over-constrained problems using QuickXPlain with speculative executions. J. Intell. Inf. Syst. 57 (3), 491–508. http://dx.doi.org/10.1007/s10844-021-00675-4.

Wohlin, C., Runeson, P., Höst, M., Ohlsson, M.C., Regnell, B., 2012. Experimentation in Software Engineering. Springer, http://dx.doi.org/10.1007/978-3-642-29044-2.

Young, T.J., 2005. Using Aspectj to Build a Software Product Line for Mobile Devices (Ph.D. thesis). In: Retrospective Theses and Dissertations, 1919-2007, University of British Columbia, http://dx.doi.org/10.14288/1.0051632, URL https://open.library.ubc.ca/collections/ubctheses/831/items/1.0051632.

José Miguel Horcas is an Associate Professor in the Department of Languages and Computer Science at the Universidad de Málaga (UMA). He earned his M.Sc. in Computer Science in 2012 and his Ph.D. in 2018, both from UMA. He is a member of the CAOSD research group in the Software Engineering and Technology Research Institute (ITIS) of UMA and a collaborator of the DiversoLab research group of the Universidad de Sevilla. His research focuses on software product lines (SPLs), variability, configuration, and quality attributes.

Joaquín Ballesteros received his M.Sc. degree in Computer Science from the University of Malaga, Málaga, Spain, in 2011, and his Ph.D. in Telecommunication Engineering from the same university in 2017. He is currently an Associate Professor with the Department of Computer Science and Programming Languages. His research focuses mainly in assistive robotics, its self-adaptation and applied AI.

Mónica Pinto received her M.Sc. degree and Ph.D. in Computer Science from the University of Málaga, Spain. She has been an Associate Professor since 2009 with the Languages and Computer Science Department of the University of Málaga. She is a member of the CAOSD research group and of the Software Engineering and Technology Research Institute (ITIS) of the University of Málaga. She actively participates in Spanish and European research projects. Her main research areas are energy-aware software development, quality-driven variability modeling and analysis, model-driven software engineering, and Edge computing systems development. She has served on the organization committee of conferences such as AOSD, Modularity, ICSE, EASE, and SPLC, playing different roles.

Prof. Lidia Fuentes earned her M.Sc. and Ph.D. in Computer Science from the Universidad de Málaga. Since 1993, she has been teaching at the Department of Lenguajes y Ciencias de la Computación, where she became the first female Full Professor. Prof. Fuentes heads the CAOSD research group (http://caosd.lcc.uma.es) and has co-authored over three hundred publications on Software Engineering techniques like variability and objective models applied to Cyber-physical systems and Mobile Networks. She has a strong international presence, leading European projects and serving on program committees for renowned international conferences, like SPLC, CAiSE, VaMoS or ECOOP.