Lazy product discovery in huge configuration spaces

Michael Lienhardt, Ferruccio Damiani, Einar Broch Johnsen, Jacopo Mauro

Lazy Product Discovery in Huge Configuration Spaces

Michael Lienhardt
ONERA – The French Aerospace Lab
France
michael.lienhardt@onera.fr

Ferruccio Damiani
University of Turin
Italy
ferruccio.damiani@unito.it

Einar Broch Johnsen
University of Oslo
Norway
einarj@ifi.uio.no

Jacopo Mauro
University of Southern Denmark
Denmark
mauro@sdu.dk

ABSTRACT

Highly-configurable software systems can have thousands of interdependent configuration options across different subsystems. In the resulting configuration space, discovering a valid product configuration for some selected options can be complex and error prone. The configuration space can be organized using a feature model, fragmented into smaller interdependent feature models reflecting the configuration options of each subsystem.

We propose a method for lazy product discovery in large fragmented feature models with interdependent features. We formalize the method and prove its soundness and completeness. The evaluation explores an industrial-size configuration space. The results show that lazy product discovery has significant performance benefits compared to standard product discovery, which in contrast to our method requires all fragments to be composed to analyze the feature model. Furthermore, the method succeeds when more efficient, heuristics-based engines fail to find a valid configuration.

CCS CONCEPTS

• **Software and its engineering → Software product lines**; *Feature interaction*; *Abstraction, modeling and modularity*; *Software libraries and repositories*; *Software creation and management*;

KEYWORDS

Software Product Lines, Configurable Software, Variability Modeling, Feature Models, Composition, Linux Distribution

ACM Reference Format:
Michael Lienhardt, Ferruccio Damiani, Einar Broch Johnsen, and Jacopo Mauro. 2020. Lazy Product Discovery in Huge Configuration Spaces. In *42nd International Conference on Software Engineering (ICSE '20), May 23–29, 2020, Seoul, Republic of Korea.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3377811.3380372

ICSE '20, May 23–29, 2020, Seoul, Republic of Korea

ACM ISBN 978-1-4503-7121-6/20/05…$15.00
https://doi.org/10.1145/3377811.3380372

1 INTRODUCTION

Highly-configurable software systems can have thousands of interdependent configuration options across different subsystems. In the resulting configuration space, different software variants can be obtained by selecting among these configuration options. The interdependencies between options are typically caused by interaction in the resulting software system. Constructing a well-functioning software variant can be a complex and error-prone process [7].

Feature models [8] allow us to organize the configuration space and facilitate the construction of software variants by describing configuration options using interdependent *features* [32]: a feature is a name representing some functionality, a set of features is called a *configuration*, and each software variant is identified by a *valid configuration* (called a *product*, for short).

Highly-configurable software systems can consist of thousands of features and combine several subsystems [12, 13, 37, 56], each with different features. The construction and maintenance of feature models with thousands of features for such highly-configurable systems, can be simplified by representing large feature models as sets of smaller interdependent feature models [12, 49] which we call *fragments*. However, the analysis of such fragmented feature models usually requires the fragments to be composed, to enable the application of existing analysis techniques [9, 10, 43, 53, 58, 59]. To this aim, many approaches for composing feature models from fragments have been investigated [3, 6, 14, 16, 48, 52].

The analysis of fragmented feature models can be simplified if suitable abstractions can safely replace some of the feature model fragments in the analysis. This simplification can be realized by means of *feature-model interfaces* [51]. A feature-model interface is a feature model that hides some of the features and dependencies of another feature model (thus, interfaces are closely related to *feature-model slicing* [4]). An interface can be used instead of a feature model fragment to simplify the overall feature model. For certain analyses, working on the simplified feature model produces results that also hold for the original feature model and for any feature model where the interface is replaced by a fragment compatible with the interface.

This paper addresses automated *product discovery* in large configuration spaces represented as sets of interdependent feature models. Product discovery (sometimes called product configuration) is a particular analysis for finding a product which includes a desired set of features [26]. We aim at automatically discovering a product

that contains a given set of features from the feature model fragments, without having to compose all the fragments to apply the analysis. This work is motivated by our recent experiences in applying techniques for variability modeling to automated product discovery in industrial use cases such as Gentoo [23], a source-based Linux distribution that consists of many highly-configurable packages. The March 1st 2019 version of the Gentoo distribution comprises 671617 features spread across 36197 feature models. Gentoo's huge configuration space can be seen as the composition of the feature models for all its packages, where package interdependencies are modeled as shared features. Gentoo's official package manager and distribution system Portage [24] achieves (via its emerge tool) efficiency at the expense of completeness; i.e., in some cases this tool fails to discover a product that contains a given set of features, although such a product exists. We show that feature model interfaces [51], which were developed to support analysis reuse for feature model evolution in fragmented feature models, do not allow us to reach our aim of complete and efficient automated product discovery.

We propose a novel method for product discovery in sets of interdependent feature models. The proposed method is lazy in the sense that features are added incrementally to the analysis until a product is found. We provide a formal account of the method and evaluate it by implementing an efficient and complete dependency solver for Gentoo. In short, our contributions are:

(1) we strengthen feature model interfaces to enable lazy product discovery in sets of interdependent feature models;
(2) we propose an efficient and complete algorithm for lazy product discovery in huge configuration spaces;
(3) we provide an open-source implementation of the proposed algorithm;[1] and
(4) we evaluate the potential of lazy product discovery in terms of experiments on an industrial-size configuration space.[2]

2 MOTIVATION AND OVERALL CONCEPT

A software system like the Gentoo distribution comprises 36197 configurable packages, as of its March 1st 2019 version. The configuration space of each package can be represented by a feature model; the overall configuration space of Gentoo can then be represented by a feature model that is the composition of the feature models of the 36197 packages. The resulting feature model has 671617 features, and thus a configuration space with up to 2^{671617} solutions.

Gentoo's official package manager Portage implements an optimized, heuristics-based product-discovery algorithm to find products in this configuration space. This algorithm is not complete; i.e., it fails to solve some product-discovery problems that have solutions. To the best of our knowledge, existing complete product-discovery approaches need to load the entire feature model to find products. Consequently, they do not scale to product-discovery problems of the size of Gentoo's configuration space.

[1] The lazy product-discovery tool is available at https://github.com/gzoumix/pdepa and at archive.softwareheritage.org/browse/origin/https://github.com/gzoumix/pdepa.git
[2] The evaluation artifact is available at https://doi.org/10.6084/m9.figshare.11728914.v4 and https://doi.org/10.5281/zenodo.3633643

Listing 1: Lazy product-discovery algorithm

```
1  input S: set of feature models
2  input c: configuration
3  var Y = c
4  var M' = compose({pick_cut(M, Y)|M ∈ S})
5  var solution = select(M', c)
6  while (solution ≠ None ∧ solution ⊈ Y):
7    Y = Y ∪ solution
8    M' = compose({pick_cut(M, Y)|M ∈ S})
9    solution = select(M', c)
10 return solution
```

In this paper we target product discovery in huge configuration spaces, such as for Gentoo, that can be described by a feature model represented as a set S of feature models with shared features, where loading the overall feature model (i.e., the whole set S) is too expensive. We propose *lazy product discovery*, a product-discovery method that loads the elements of S incrementally, until it finds a product of the overall feature model. The method relies on the notion of a *cut of a feature model $\mathcal{M}$ for a set of features Y*. This is a feature model $\mathcal{M}'$ whose products are products of $\mathcal{M}$ and include all the products of $\mathcal{M}$ that contain a feature in Y.

The proposed algorithm, shown in Listing 1, takes as input a set S of feature models with shared features and a set c of features to be included in the discovered product. After initialization, the algorithm incrementally loads cuts until a solution has been found. Let $\mathcal{M}_0$ denote the composition of the feature models in S. The algorithm returns a (not necessarily minimal) product of $\mathcal{M}_0$ which includes all features in c, whenever such a product exists; otherwise, it returns the special value None. The algorithm relies on the following three auxiliary functions:

(1) `pick_cut`$(\mathcal{M}, Y)$: a function that, given a feature model $\mathcal{M}$ and a set of features Y, returns a cut of $\mathcal{M}$ for Y;
(2) `compose`$(\{\mathcal{M}_1, \ldots, \mathcal{M}_n\})$: a function that, given a set of feature models $\mathcal{M}_1, \ldots, \mathcal{M}_n$, returns the composition of the feature models in the set; and
(3) `select`$(\mathcal{M}, c)$: a function that, given a feature model $\mathcal{M}$ and a set of features c, returns a product of $\mathcal{M}$ containing all the features in c if it exists, and **None** otherwise.

Assuming that the auxiliary functions (1), (2) and (3) work, we have that on Line 6 the following loop invariants hold:

Inv1: $c \subseteq Y$.
Inv2: solution is a product of $\mathcal{M}'$ which includes all features in c, whenever such a product exists; otherwise solution is the special value **None**.
Inv3: if solution is a product of $\mathcal{M}'$ and solution $\subseteq Y$, then solution is also a product of $\mathcal{M}_0$.
Inv4: If $\mathcal{M}'$ has no product which includes all features in c, then neither does $\mathcal{M}_0$.

Checking that **Inv1** holds is straightforward: just observe that on Line 3 the variable Y is initialized to c and that at each iteration of the **while** loop new features are added to Y on Line 7. Checking that **Inv2** holds is equally straightforward: according to the description of the auxiliary functions (1), (2) and (3), the invariant

is established on Lines 4 and 5 as well as on Lines 8 and 9. The fact that **Inv3** and **Inv4** hold is shown in the proof of Theorem 4 in Section 5. The algorithm terminates because at each iteration of the **while** loop, the size of the set Y (which, by construction, only contains features from the features models in S) increases. When the algorithm terminates we have that either solution = **None** or **None** $\neq$ solution $\subseteq Y$. In the first case (by **Inv4**) we have that $\mathcal{M}_0$ has no product that contains all the features in c, while in the second case (by **Inv3**) we have that solution is a product of $\mathcal{M}_0$ that contains all the features in c.

The laziness of this algorithm stems from the fact that it does not need to consider $\mathcal{M}_0$ at once. Instead, the algorithm starts by considering the composition of the cuts of the feature models for $Y = c$ and then iterates by considering bigger and bigger cuts until the candidate solution is contained in the set Y. When this happens we know, for the properties of the cut, that the found solution is also a solution for $\mathcal{M}_0$.

The algorithm's efficiency in finding a product with the features in c (see Lines 4, 5 and 8, 9 of Listing 1) compared to executing $\texttt{select}(\mathcal{M}_0, c)$, depends on the degree to which the feature models in S are such that:

- computing $\texttt{pick\_cut}(\mathcal{M}, Y)$ is efficient,
- the feature models $\mathcal{M}'$ are small compared to $\mathcal{M}_0$,
- $\texttt{select}(\mathcal{M}', c)$ performs better than $\texttt{select}(\mathcal{M}_0, c)$, and
- a small number of iterations of the **while**-loop is required.

For the Gentoo distribution, each feature model $\mathcal{M}_i$ in S has a distinguished feature f_i such that the constraints expressed by $\mathcal{M}_i$ are enabled only if f_i is selected (see Section 6.1). This reflects that each $\mathcal{M}_i$ corresponds to a Gentoo package that is installed if and only if f_i is selected. Therefore, the function $\texttt{pick\_cut}(\mathcal{M}, Y)$ can be efficiently implemented by returning $\mathcal{M}_i$ if $f_i \in Y$, and by returning a feature model that expresses no constraints (and can, therefore, be ignored by the composition that builds $\mathcal{M}'$) otherwise.

The rest of this paper is organized as follows: Sections 3–5 provide a formal account of the lazy product-discovery method that culminates in the proof that **Inv3** and **Inv4** hold, Section 6 evaluates the performance of the lazy product-discovery algorithm by means of experiments, and Sections 7 and 8 discuss related work and conclude the paper, respectively.

3 A FORMALIZATION OF FEATURE MODELS

This section presents a formalization of feature models (FM) and related notions, including feature model interfaces and composition.

3.1 Feature Model Representations

Different representations of feature models are discussed, e.g., by Batory [8]. In this paper, we will rely on the *propositional formula* representation of feature models. In this representation, a feature model is given by a pair $(\mathcal{F}, \phi)$ where:

- $\mathcal{F}$ is a set of features, and
- ϕ is a propositional formula where the variables x are feature names: $\phi ::= x \mid \phi \wedge \phi \mid \phi \vee \phi \mid \phi \rightarrow \phi \mid \neg\phi$.

A propositional formula ϕ over a set of features $\mathcal{F}$ represents the feature models whose products are configurations $\{x_1, ..., x_n\} \subseteq \mathcal{F}$ $(n \geq 0)$ such that ϕ is satisfied by assigning value true to the variables x_i $(1 \leq i \leq n)$ and false to all other variables.

Example 1 (A propositional representation of glibc FM). *Gentoo packages can be configured by selecting features (called* use flags *in Gentoo), which may trigger dependencies or conflicts between packages. Version 2.29 of the* glibc *library, that contains the core functionalities of most Linux systems, is provided by the package* sys-libs/glibc-2.29-r2 *(abbreviated to* glibc *in the sequel). This package has many dependencies, including (as expressed in Gentoo's notation):*

```
doc? ( sys−apps/texinfo )
    vanilla?( !sys−libs/timezone−data )
```

This dependency expresses that glibc *requires the* texinfo *documentation generator (provided by any version of the* sys-apps/texinfo *package) whenever the feature* doc *is selected and if the feature* vanilla *is selected, then* glibc *conflicts with any version of the time zone database (as stated with the* !sys-libs/timezone-data *constraint). These* dependencies *and* conflicts *can be expressed by a feature model* $(\mathcal{F}_{\text{glibc}}, \phi_{\text{glibc}})$ *where*

$$\mathcal{F}_{\text{glibc}} = \{\text{glibc, txinfo, tzdata, glibc:doc, glibc:v}\}, \text{ and}$$
$$\phi_{\text{glibc}} = \text{glibc} \rightarrow ((\text{glibc:doc} \rightarrow \text{txinfo}) \wedge (\text{glibc:v} \rightarrow (\neg\text{tzdata})).$$

Here, the feature glibc *represents the* glibc *package;* txinfo *represents any* sys-apps/texinfo *package;* tzdata *represents any version of the* sys-libs/timezone-data *package; and* glibc:doc *(resp.* glibc:v*) represents the* glibc*'s* doc *(resp.* vanilla*) use flag.*

The propositional representation of feature models works well in practice [9, 44, 58] and we shall use it for the evaluation of the proposed method (in Section 6). In contrast, to simplify the proofs, we follow Schröter *et al.* [51] in using an extensional representation of feature models to present our theory.

Definition 1 (Feature model, extensional representation). *A* Feature Model $\mathcal{M}$ *is a pair* $(\mathcal{F}, \mathcal{P})$ *where* $\mathcal{F}$ *is a set of features and* $\mathcal{P} \subseteq 2^{\mathcal{F}}$ *a set of products.*

Example 2 (An extensional representation of glibc FM). *Let* 2^X *denote the powerset of* X*. The feature model of Example 1 can be given an extensional representation* $\mathcal{M}_{\text{glibc}} = (\mathcal{F}_{\text{glibc}}, \mathcal{P}_{\text{glibc}})$ *where* $\mathcal{F}_{\text{glibc}}$ *is the same as in Example 1 and*

$$\begin{aligned}\mathcal{P}_{\text{glibc}} = &\{\{\text{glibc}\}, \{\text{glibc, txinfo}\}, \{\text{glibc, tzdata}\}, \{\text{glibc, txinfo, tzdata}\}\} \cup \\ &\{\{\text{glibc, glibc:doc, txinfo}\}, \{\text{glibc, glibc:doc, txinfo, tzdata}\}\} \cup \\ &\{\{\text{glibc, glibc:v}\}, \{\text{glibc, glibc:v, txinfo}\}\} \cup \\ &\{\{\text{glibc, glibc:doc, glibc:v, txinfo}\}\} \cup \\ &2^{\{\text{txinfo, tzdata, glibc:doc, glibc:v}\}}.\end{aligned}$$

In the description of $\mathcal{P}_{\text{glibc}}$*, the first line contains products with* glibc *but none of its use flags are selected, so* texinfo *and* tzdata *can be freely installed; the second line contains products with the use flag* doc *selected in* glibc*, so a package of* sys-apps/texinfo *is always required; the third line contains products with the use flag* vanilla *selected in* glibc*, so no package of* sys-libs/timezone-data *is allowed; the forth line contains products with both* glibc*'s use flags selected,*

so sys-apps/texinfo *is mandatory and* sys-libs/timezone-data *forbidden; finally, the fifth line represents products without* glibc*, so all combinations of other features are possible, including the empty set.*

Definition 2 (Empty FM, void FMs, and pre-products). *The* empty *feature model, denoted* $\mathcal{M}_\emptyset = (\emptyset, \{\emptyset\})$*, has no features and has just the empty product* $\emptyset$*. A* void *feature model is a feature model that has no products, i.e., it has the form* $(\mathcal{F}, \emptyset)$ *for some* $\mathcal{F}$*. A* pre-product *of a feature model* $\mathcal{M}$ *is a configuration* c *that can be extended to a product of* $\mathcal{M}$ *(more formally,* $c \subseteq p$ *for some product* p *of* $\mathcal{M}$*).*

Based on the above definition of a pre-product, we identify two related search problems.

Definition 3 (Feature compatibility, product discovery). *Consider a feature model* $\mathcal{M}$ *and a set of features* c *in* $\mathcal{M}$*. The* feature-compatibility problem *for* c *in* $\mathcal{M}$ *is the problem of determining whether* c *is a pre-product of* $\mathcal{M}$ *(i.e., whether the features in* c *are compatible with the products in* $\mathcal{M}$*). The* product-discovery problem *for* c *in* $\mathcal{M}$ *is the problem of finding a product of* $\mathcal{M}$ *that extends* c*.*

Clearly, the feature-compatibility problem for c in $\mathcal{M}$ has a positive answer if and only if the product-discovery problem for c in $\mathcal{M}$ has a solution.

3.2 Feature Model Interfaces

Feature model interfaces were defined by Schröter *et al.* [51] as a binary relation $\preceq$, expressing that a feature model $\mathcal{M}'$ is an interface of a feature model $\mathcal{M}$ if $\mathcal{M}'$ ignores some features of $\mathcal{M}$.

Definition 4 (FM interface relation). *A feature model* $\mathcal{M}' = (\mathcal{F}', \mathcal{P}')$ *is an* interface *of feature model* $\mathcal{M} = (\mathcal{F}, \mathcal{P})$*, denoted as* $\mathcal{M}' \preceq \mathcal{M}$*, iff* $\mathcal{F}' \subseteq \mathcal{F}$ *and* $\mathcal{P}' = \{p \cap \mathcal{F}' \mid p \in \mathcal{P}\}$*.*

Note that, for all feature models $\mathcal{M}' = (\mathcal{F}', \mathcal{P}')$ and $\mathcal{M}$, if $\mathcal{M}' \preceq \mathcal{M}$ then (i) all products of $\mathcal{M}'$ are pre-products of $\mathcal{M}$ and (ii) $\mathcal{M}'$ is the only interface of $\mathcal{M}$ which has exactly the features $\mathcal{F}'$ (i.e., $\mathcal{M}'$ is completely determined by $\mathcal{F}'$).

Example 3 (An interface for glibc FM). *The feature model*

$$\mathcal{F} = \{\text{glibc, glibc:v}\}$$
$$\mathcal{P} = \{\emptyset, \{\text{glibc}\}, \{\text{glibc, glibc:v}\}\}$$

is the interface of the feature model $\mathcal{M}_{\text{glibc}}$ *from Example 2 that is determined by the features* glibc *and* glibc:v*.*

The interface relation for feature models is a *partial order* (i.e., it is reflexive, transitive and anti-symmetric) and the empty feature model $\mathcal{M}_\emptyset$ is an interface of every non-void feature model $\mathcal{M}$. Moreover, $\mathcal{M}$ is void if and only if $(\emptyset, \emptyset) \preceq \mathcal{M}$.

The notion of a feature model interface is closely related to that of a *feature model slice*, which was defined by Acher *et al.* [4] as a unary operator Π_Y restricting a feature model to a set Y of features. Given a feature model $\mathcal{M}$, $\Pi_Y(\mathcal{M})$ is the feature model obtained from $\mathcal{M}$ by removing the features not in Y.

Definition 5 (FM slice operator). *The* slice operator Π_Y *on feature models, where* Y *is a set of features, is defined by:*

$$\Pi_Y((\mathcal{F}, \mathcal{P})) = (\mathcal{F} \cap Y, \{p \cap Y \mid p \in \mathcal{P}\}).$$

Note that, for every feature model $\mathcal{M} = (\mathcal{F}, \mathcal{P})$ and set of features Y, the feature model $\Pi_Y(\mathcal{M}) = (\mathcal{F}', \mathcal{P}')$ is the unique interface of $\mathcal{M}$ such that $\mathcal{F}' = \mathcal{F} \cap Y$. Moreover, for every interface $\mathcal{M}_1 = (\mathcal{F}_1, \mathcal{P}_1)$ of $\mathcal{M}$ it holds that $\mathcal{M}_1 = \Pi_{\mathcal{F}_1}(\mathcal{M})$.

Example 4 (A slice of glibc FM). *The feature model interface in Example 3 can be obtained by applying* $\Pi_{\{\text{glibc, glibc:v}\}}$ *to the feature model* $\mathcal{M}_{\text{glibc}}$ *of Example 2.*

3.3 Feature Model Composition

Highly-configurable software systems often consist of many interdependent, configurable packages [23, 37, 38]. The variability constraints of each of these packages can be represented by a feature model. Therefore, configuring two (or more packages) in such a way that they can be installed together corresponds to identifying a product in a suitable *composition* of their associated feature models. In the propositional representation of feature models, such composition corresponds to logical conjunction; i.e., the composition of two feature models $(\mathcal{F}_1, \phi_1)$ and $(\mathcal{F}_2, \phi_2)$ is the feature model

$$(\mathcal{F}_1 \cup \mathcal{F}_2, \phi_1 \wedge \phi_2).$$

In the extensional representation of feature models, this form of composition corresponds to the binary operator $\bullet$ of Schröter *et al.* [51], which is similar to the join operator from relational algebra [17].

Definition 6 (FM composition). *The composition of two feature models* $\mathcal{M}_1 = (\mathcal{F}_1, \mathcal{P}_1)$ *and* $\mathcal{M}_2 = (\mathcal{F}_2, \mathcal{P}_2)$*, denoted* $\mathcal{M}_1 \bullet \mathcal{M}_2$*, is the feature model defined by:*

$$\mathcal{M}_1 \bullet \mathcal{M}_2 = (\mathcal{F}_1 \cup \mathcal{F}_2, \{p \cup q \mid p \in \mathcal{P}_1, q \in \mathcal{P}_2, p \cap \mathcal{F}_2 = q \cap \mathcal{F}_1\}).$$

The composition operator $\bullet$ is associative and commutative, with $\mathcal{M}_\emptyset$ as identity element (i.e., $\mathcal{M} \bullet \mathcal{M}_\emptyset = \mathcal{M}$). Composing a feature model with a void feature model yields a void feature model: $(\mathcal{F}_1, \mathcal{P}_1) \bullet (\mathcal{F}_2, \emptyset) = (\mathcal{F}_1 \cup \mathcal{F}_2, \emptyset)$.

Example 5 (Composing glibc and gnome-shell FMs). *Let us consider another important package of the Gentoo distribution:* gnome-shell*, a core component of the Gnome Desktop environment. Version 3.30.2 of* gnome-shell *is provided by the package* gnome-base/gnome-shell-3.30.2-r2 *(abbreviated to* g-shell *in the sequel), and its dependencies include the following statement:*

networkmanager?(sys−libs/timezone−data).

This dependency expresses that g-shell *requires any version of the time zone database when the feature* networkmanager *is selected.*

The propositional representation *of this dependency can be captured by the feature model* $(\mathcal{F}_{\text{g-shell}}, \phi_{\text{g-shell}})$*, where*

$$\mathcal{F}_{\text{g-shell}} = \{\text{g-shell, tzdata, g-shell:nm}\}, \text{ and}$$
$$\phi_{\text{g-shell}} = \text{g-shell} \rightarrow (\text{g-shell:nm} \rightarrow \text{tzdata}).$$

The corresponding extensional representation *of this feature model is* $\mathcal{M}_{\text{g-shell}} = (\mathcal{F}_{\text{g-shell}}, \mathcal{P}_{\text{g-shell}})$*, where:*

$$\mathcal{P}_{\text{g-shell}} = \{\{\text{g-shell}\}, \{\text{g-shell, tzdata}\}\} \cup$$
$$\{\{\text{g-shell, tzdata, g-shell:nm}\}\} \cup$$
$$2^{\{\text{tzdata, g-shell:nm}\}}.$$

Here, the first line contains products with g-shell *but none of its use flags are selected:* tzdata *can be freely selected; the second line*

is the product where g-shell:nm *is also selected and* tzdata *becomes mandatory; finally, the third line represents products without* g-shell.

The propositional representation *of the composition is the feature model* $(\mathcal{F}_{\text{full}}, \phi_{\text{full}})$*, where*

$$\begin{aligned}\mathcal{F}_{\text{full}} &= \mathcal{F}_{\text{glibc}} \cup \mathcal{F}_{\text{g-shell}}\\ &= \{\text{glibc, txinfo, tzdata, g-shell, glibc:doc, glibc:v, g-shell:nm}\},\ \textit{and}\\ \phi_{\text{full}} &= \phi_{\text{glibc}} \wedge \phi_{\text{g-shell}}\\ &= (\text{glibc} \rightarrow ((\text{glibc:doc} \rightarrow \text{txinfo}) \wedge (\text{glibc:v} \rightarrow (\neg\text{tzdata})))\wedge\\ &\quad (\text{g-shell} \rightarrow (\text{g-shell:nm} \rightarrow \text{tzdata})).\end{aligned}$$

The extensional representation *of the composition is the feature model* $\mathcal{M}_{\text{full}} = \mathcal{M}_{\text{glibc}} \bullet \mathcal{M}_{\text{g-shell}} = (\mathcal{F}_{\text{full}}, \mathcal{P}_{\text{full}})$ *where*

$$\begin{aligned}\mathcal{P}_{\text{full}} = \mathcal{P}_{\text{glibc}} \cup \mathcal{P}_{\text{g-shell}} \cup 2^{\{\text{txinfo, tzdata, glibc:doc, glibc:v, g-shell:nm}\}} \cup\\ \{\{\text{glibc, g-shell}\} \cup p \mid p \in 2^{\{\text{txinfo, tzdata}\}}\} \cup\\ \{\{\text{glibc, glibc:doc, txinfo, g-shell}\} \cup p \mid p \in 2^{\{\text{tzdata}\}}\} \cup\\ \{\{\text{glibc, glibc:v, g-shell}\} \cup p \mid p \in 2^{\{\text{txinfo}\}}\} \cup\\ \{\{\text{glibc, g-shell, g-shell:nm, tzdata}\} \cup p \mid p \in 2^{\{\text{txinfo}\}}\} \cup\\ \{\{\text{glibc, glibc:doc, glibc:v, txinfo, g-shell}\}\} \cup\\ \{\{\text{glibc, glibc:doc, txinfo, g-shell, g-shell:nm, tzdata}\}\}.\end{aligned}$$

Here, the first line contains the products where glibc *and* g-shell *do not interact, i.e., either when they are not installed, or only one of them is installed; the second line contains the products where both* glibc *and* g-shell *are installed, but without use flags selected, so all optional package can be freely selected; the third line contains the products with the* glibc*'s use flag* doc *selected, so* sys-apps/texinfo *becomes mandatory; the fourth line contains the products with the* glibc*'s use flag* vanilla *selected, so* sys-libs/timezone-data *is forbidden; the fifth line contains the products with the* g-shell*'s use flag* vanilla *network manager, so* sys-libs/timezone-data *is mandatory; the sixth line contains the product with* glibc*'s both use flags selected and the seventh line contains the product with* glibc*'s use flag* doc *and* g-shell*'s use flag* networkmanager *are selected.*

4 PROBLEM STATEMENT

Many case studies show that the size of feature models used to model real configuration spaces can be challenging for both humans and machines [12, 51, 55, 59], including the feature model for the source-based Linux distribution Gentoo [23] mentioned above. The state-of-the-art strategy used to address this challenge is to represent large feature models by sets of smaller interdependent feature models [12, 49]. The resulting interdependencies between different feature models can be expressed using shared features [51].

The feature compatibility problem for a given set of features (see Definition 3) can be decided without first composing the considered feature models when the feature models are disjoint, as it suffices to inspect each feature model independently. Namely, feature-model slices can be used to formulate a *feature-compatibility criterion* for the case with no shared features between the feature models, as shown by the following theorem:

Theorem 1 (Feature-compatibility criterion for disjoint FMs). *Consider the feature models* $\mathcal{M}_i = (\mathcal{F}_i, \mathcal{P}_i)$ $(1 \le i \le n)$ *with pairwise no shared features (i.e.,* $1 \le i \ne j \le n$ *implies* $\mathcal{F}_i \cap \mathcal{F}_j = \emptyset$*). Then a configuration* c *is a pre-product of the feature model* $\mathcal{M} = \bullet_{1\le i\le n}\mathcal{M}_i$ *if and only if* c *is a subset of* $\bigcup_{1\le i\le n}\mathcal{F}_i$ *and for all* $\mathcal{M}_i$ *the configuration* $c \cap \mathcal{F}_i$ *is a product of* $\Pi_c(\mathcal{M}_i)$*.*

Proof. Let $\mathcal{M} = (\mathcal{F}, \mathcal{P})$.

Case $\Rightarrow$. Since c is a pre-product of $\mathcal{M}$, by definition there exist $p \in \mathcal{P}$ such that $c \subseteq p$. Hence $c \subseteq \mathcal{F} = \bigcup_{1\le i\le n}\mathcal{F}_i$. Let now consider $\Pi_c(\mathcal{M}_i)$ for any $1 \le i \le n$: by definition $p \cap c \cap \mathcal{F}_i$ is a product of this feature model, and by construction, $p \cap c \cap \mathcal{F}_i = c \cap \mathcal{F}_i$. Hence, $c \cap \mathcal{F}_i$ is a product of $\Pi_c(\mathcal{M}_i)$ for any $1 \le i \le n$.

Case $\Leftarrow$. Since for any $1 \le i \le n$, $c \cap \mathcal{F}_i$ is a product of $\Pi_c(\mathcal{M}_i)$, there exist $p_i \in \mathcal{P}_i$ such that $c \cap \mathcal{F}_i = p_i \cap c$. Let consider the configuration $p = \bigcup_{1\le i\le n} p_i$. Since the feature models $\mathcal{M}_i$ do not share features, we have $p_i \cap \mathcal{F}_j = \emptyset = p_j \cap \mathcal{F}_i$ for all $1 \le i \ne j \le n$. Hence p is a product of $\mathcal{M}$. Moreover, we have that:

$$p \cap c = \bigcup_{1\le i\le n}(p_i \cap c) = \bigcup_{1\le i\le n}(c \cap \mathcal{F}_i) = c \cap \bigcup_{1\le i\le n}\mathcal{F}_i = c.$$

Hence $c \subseteq p$ holds, which means that c is a pre-product of $\mathcal{M}$. □

Unfortunately, the feature compatibility criterion of Theorem 1 does not work for feature models with shared features. The problem can be illustrated by the following example.

Example 6 (Feature compatibility with shared features). *Consider the two feature models* $\mathcal{M}_{\text{glibc}}$ *and* $\mathcal{M}_{\text{g-shell}}$ *from Examples 2 and 5, and the configuration* $c = \{\text{glibc, glibc:v, g-shell, g-shell:nm}\}$*. We have*

$$\Pi_c(\mathcal{M}_{\text{glibc}}) = (\{\text{glibc, glibc:v}\}, 2^{\{\text{glibc, glibc:v}\}}),\ \textit{and}$$
$$\Pi_c(\mathcal{M}_{\text{g-shell}}) = (\{\text{g-shell, g-shell:nm}\}, 2^{\{\text{g-shell, g-shell:nm}\}}).$$

Here, we have that $c \subseteq \mathcal{F}_{\text{glibc}} \cup \mathcal{F}_{\text{g-shell}}$ *and it is clear from the previous equation that* $c \cap \mathcal{F}_{\text{glibc}} = \{\text{glibc, glibc:v}\}$ *is a product of* $\Pi_c(\mathcal{M}_{\text{glibc}})$ *and that* $c \cap \mathcal{F}_{\text{g-shell}} = \{\text{g-shell, g-shell:nm}\}$ *is a product of* $\Pi_c(\mathcal{M}_{\text{g-shell}})$*. However,* c *is not a pre-product of* $\mathcal{M}_{\text{glibc}} \bullet \mathcal{M}_{\text{g-shell}}$*, since the use flag* g-shell:nm *requires a timezone database to be installed while the use flag* glibc:v *forbids it.*

In this paper we address complete and efficient product discovery in sets of interdependent feature models. To this aim, we define a novel criterion which, given some selected features, enables solving the product-discovery problem for a set of feature model fragments with shared features, without composing all the fragments.

5 LAZY PRODUCT DISCOVERY

We are looking for a product-discovery criterion which works for interdependent feature models, similar to how the feature-compatibility criterion given in Theorem 1 works for disjoint feature models. The solution lies in a novel criterion based on strengthening the feature model interfaces. Given feature models with shared features $\mathcal{M}_i = (\mathcal{F}_i, \mathcal{P}_i)$ and a set of selected features c, we need feature model interfaces $\mathcal{M}'_i$ that reflect how c is related to other features in $\mathcal{M}_i$ in order to guarantee that the interface behaves similarly to $\mathcal{M}_i$ with respect to the feature-compatibility problem for c. More formally, the interface $\mathcal{M}'_i$ must satisfy the following conditions:

(1) $\Pi_c(\mathcal{M}_i) \preceq \mathcal{M}'_i$; and
(2) the products of $\mathcal{M}'_i$ are among the products of $\mathcal{M}_i$.

Example 7 (Feature compatibility with shared features continued). *Consider feature models* $\mathcal{M}_{\text{glibc}}$ *and* $\mathcal{M}_{\text{g-shell}}$ *and configuration* c*, as discussed in Example 6. Let* $c_1 = \{\text{glibc, glibc:v}\}$ *and*

$c_2 = \{\text{glibc, tzdata, glibc:v}\}$. *We can see that the interface* $\mathcal{M}'_{\text{glibc}} = \Pi_{c_2}(\mathcal{M}_{\text{glibc}})$ *of* $\mathcal{M}_{\text{glibc}}$ *satisfies (with* $i =$ glibc*) conditions (1) and (2) above. Since* $\Pi_c(\mathcal{M}_{\text{glibc}}) = \Pi_{c_1}(\mathcal{M}_{\text{glibc}})$ *and* $c_2 \setminus c_1 = \{\text{tzdata}\}$*, this shows that it is important to consider the feature* tzdata *when checking whether c is a pre-product of a composed feature model including* $\mathcal{M}_{\text{glibc}}$.

Let us now introduce terminology for different restrictions to the interface relation that satisfy one or both of the conditions (1) and (2) given above, and investigate some of their properties.

Definition 7 (FM extended slice, conservative interface, and cut relations). *Given a set of features Y and two feature models* $\mathcal{M}' = (\mathcal{F}', \mathcal{P}')$ *and* $\mathcal{M} = (\mathcal{F}, \mathcal{P})$*, we say that*

(1) $\mathcal{M}'$ *is an* extended slice for Y *of* $\mathcal{M}$*, denoted* $\mathcal{M}' \preceq_Y \mathcal{M}$*, iff* $\Pi_Y(\mathcal{M}) \preceq \mathcal{M}' \preceq \mathcal{M}$ *holds;*
(2) $\mathcal{M}'$ *is a* conservative interface *of* $\mathcal{M}$*, denoted* $\mathcal{M}' \trianglelefteq \mathcal{M}$*, iff both* $\mathcal{M}' \preceq \mathcal{M}$ *and* $\mathcal{P}' \subseteq \mathcal{P}$ *hold; and*
(3) $\mathcal{M}'$ *is a* cut for Y *of* $\mathcal{M}$*, denoted* $\mathcal{M}' \trianglelefteq_Y \mathcal{M}$*, iff* $\mathcal{M}'$ *is both an extended slice for Y and a conservative interface.*

Note that $\trianglelefteq_\emptyset = \trianglelefteq$. The relation $\trianglelefteq$ is a partial order; the feature model $(\emptyset, \emptyset)$ is the *minimum* (i.e., the smallest w.r.t. both $\preceq$ and $\trianglelefteq$) conservative interface of every void feature model; and the empty feature model $\mathcal{M}_\emptyset$ is the minimum conservative interface of every feature model that has the empty product.

The following theorem proves, in a constructive way, the existence of the minimum cut of $\mathcal{M}$ for Y, for any feature model $\mathcal{M} = (\mathcal{F}, \mathcal{P})$. Let the *minimal products* of $\mathcal{M}$ be the products that are not included in other products, and let $Y' = (\mathcal{F} \cap Y)$ be the set of features of $\mathcal{M}$ that occur in Y. Intuitively, the *minimum cut* of $\mathcal{M}$ for Y is the feature model obtained from $(Y', \emptyset)$ by incrementally adding all the minimal products of $\mathcal{M}$ (and their features) that contain a feature occurring in the feature model, until a fixed point is reached.

Theorem 2 (Characterization of the minimum cut). *For all sets Y of features and all feature models* $\mathcal{M} = (\mathcal{F}, \mathcal{P})$*, let* $\bot^{\trianglelefteq_Y}(\mathcal{M})$ *be the minimum cut of* $\mathcal{M}$ *for Y, i.e.,*

$$\bot^{\trianglelefteq_Y}(\mathcal{M}) = \min_{\trianglelefteq}\{\mathcal{M}' \mid \mathcal{M}' \trianglelefteq_Y \mathcal{M}\}.$$

Then $\bot^{\trianglelefteq_Y}(\mathcal{M}) = f^\infty(((\mathcal{F} \cap Y), \emptyset))$*, where f is the function between feature models defined by*

$$f((\mathcal{F}_1, \mathcal{P}_1)) = (\mathcal{F}_1 \cup (\bigcup_{p \in \mathcal{P}_2} p), \mathcal{P}_1 \cup \mathcal{P}_2)$$

with $\mathcal{P}_2 = \{p \in \mathcal{P} \mid \forall p' \in \mathcal{P}, (p' \subsetneq p) \Rightarrow ((p \setminus p') \cap \mathcal{F}_1 \neq \emptyset)\}$.

Proof. Let $\mathcal{M}' = ((\mathcal{F} \cap Y), \emptyset)$ and consider the partially ordered set of feature models $(S, \leq)$, defined by

- $S = \{(\mathcal{F}'', \mathcal{P}'') \mid (\mathcal{F} \cap Y) \subseteq \mathcal{F}'' \subseteq \mathcal{F} \wedge \mathcal{P}'' \subseteq \mathcal{P}\}$, and
- $(\mathcal{F}_1, \mathcal{P}_1) \leq (\mathcal{F}_2, \mathcal{P}_2)$ iff $(\mathcal{F}_1 \subseteq \mathcal{F}_2)$ and $(\mathcal{P}_1 \subseteq \mathcal{P}_2)$.

It is straightforward to see that $(S, \leq)$ is a complete lattice (with minimum $\mathcal{M}'$ and maximum $\mathcal{M}$) and that f is monotonic increasing for $\leq$. Hence, by [33], $f^\infty(\mathcal{M}')$ exists and is the minimum fixpoint of f.

We prove that the fixpoints of f are exactly the cuts of $\mathcal{M}$ for Y. Let us first consider a feature model $\mathcal{M}_Y = (\mathcal{F}_Y, \mathcal{P}_Y)$ that is a cut of $\mathcal{M}$ for Y. Since $\mathcal{M}_Y = \Pi_{\mathcal{F}_Y}(\mathcal{M})$ and $\mathcal{P}_Y \subseteq \mathcal{P}$ for all $p \in \mathcal{P}$, we have $p \cap \mathcal{F}_Y \in \mathcal{P}$. This implies that for any $p \in \mathcal{P} \setminus \mathcal{P}_Y$, there exists $p' \in \mathcal{P}$ with $p' \subsetneq p$ such that $(p \setminus p') \cap \mathcal{F}_Y = \emptyset$. By definition, we have $f(\mathcal{M}_Y) = (\bigcup_{p \in \mathcal{P}_2} p \cup \mathcal{F}_Y, \mathcal{P}_Y \cup \mathcal{P}_2)$ with

$$\mathcal{P}_2 = \{p \in \mathcal{P} \mid \forall p' \in \mathcal{P}, (p' \subsetneq p) \Rightarrow ((p \setminus p') \cap \mathcal{F}_Y \neq \emptyset)\} \subseteq \mathcal{P}_Y.$$

Hence $f(\mathcal{M}_Y) = \mathcal{M}_Y$.

Let us now consider a feature model $\mathcal{M}'_Y = (\mathcal{F}'_Y, \mathcal{P}'_Y)$ in S such that $f(\mathcal{M}'_Y) = \mathcal{M}'_Y$. First, it is clear by construction that $\mathcal{P}'_Y \subseteq \mathcal{P}$. Moreover, if we write $\mathcal{P}' = \{p \in \mathcal{P} \mid \forall p' \in \mathcal{P} \setminus \{p\}, p' \not\subseteq p\}$, it is clear from the definition of f that $\mathcal{P}' \subseteq \mathcal{P}'_Y$. Suppose that the set $M = \{p \in \mathcal{P} \mid p \cap \mathcal{F}'_Y \notin \mathcal{P}'_Y\}$ is not empty and consider p_1 a minimal element of M w.r.t. $\subseteq$. Since $p_1 \notin \mathcal{F}'_Y$, by definition of $\mathcal{P}'$, the set $N = \{p' \in \mathcal{P}'_Y \mid p' \subseteq p_1\}$ is not empty. Consider any maximal element p_2 of N w.r.t. $\subseteq$. Since $p_1 \cap \mathcal{F}'_Y \notin \mathcal{P}'_Y$, we have $(p_1 \setminus p_2) \cap \mathcal{F}'_Y \neq \emptyset$, and so the condition $\forall p' \in \mathcal{P}, (p' \subsetneq p_1) \Rightarrow ((p_1 \setminus p') \cap \mathcal{F}'_Y \neq \emptyset)$ holds. It follows that $\mathcal{M}'_Y$ is not a fixpoint of f (since applying f to $\mathcal{M}'_Y$ would add the product p_1), which contradicts the hypothesis. Hence for all $p \in \mathcal{P}$, $p \cap \mathcal{F}'_Y \in \mathcal{P}'_Y$, this means that $\mathcal{M}'_Y = \Pi_{\mathcal{F}'_Y}(\mathcal{M})$. Since by construction $Y \cap \mathcal{F} \subseteq \mathcal{F}'_Y$, we have $\Pi_Y(\mathcal{M}) \preceq \mathcal{M}'_Y \preceq \mathcal{M}$: $\mathcal{M}'_Y$ is a cut of $\mathcal{M}$ for Y.

To conclude, observe that the orders $\preceq$ and $\leq$ are equal on the set of cuts of $\mathcal{M}$ for Y. Since $f(\mathcal{M}')$ is the minimum fixpoint of f w.r.t. $\leq$, it is also the minimum cut of $\mathcal{M}$ for Y. □

Example 8 (A minimum cut of glibc FM). *Consider the feature model* $\mathcal{M}_{\text{glibc}}$ *of Example 2 and* $Y = \{\text{glibc, glibc:doc}\}$*. The minimal cut* $\bot^{\trianglelefteq_Y}(\mathcal{M}_{\text{glibc}})$ *can be computed by starting with the feature model* $(Y, \emptyset)$ *and then applying f. In the first application of f, the set* $\mathcal{P}_2$ *collects the products* $\emptyset$, {glibc}, {glibc:doc}*, and* {glibc,glibc:doc,txinfo}*. The set* $\mathcal{F}_1$ *after the first application becomes* {glibc,glibc:doc,txinfo} *and therefore, in the second application of f, the products* {txinfo}, {glibc, txinfo}*, and* {glibc:doc, txinfo} *are added to* $\mathcal{P}_2$*. At this point, further applications of f do not add further products.*

In this case, the minimum cut $\bot^{\trianglelefteq_Y}(\mathcal{M}_{\text{glibc}})$ *is different from the slice* $\Pi_Y(\mathcal{M}_{\text{glibc}})$*, since the cut keeps the information that when* glibc *and* glibc:doc *are selected, then* txinfo *also has to be selected.*

The following theorem proves sufficient criteria to guarantee that a product of the composition of cuts is also a product of the composition of the original feature models and, conversely, that the original feature model does not have a product that contains a given set of features. Intuitively, given a set of features Y and a product p of the composition of cuts for Y, if p is a subset of Y we have that p is also a product of the composition of the original feature models. Moreover, if the composition of cuts for Y has no products with the features in a set $c \subseteq Y$, then neither does the the original feature model.

Theorem 3 (Product-discovery criterion for interdependent FMs). *Consider a set Y of features, a finite set I of indices, and two sets of feature models* $\{\mathcal{M}_i = (\mathcal{F}_i, \mathcal{P}_i) \mid i \in I\}$ *and* $\{\mathcal{M}'_i = (\mathcal{F}'_i, \mathcal{P}'_i) \mid i \in I\}$ *such that for all* $i \in I$, $\mathcal{M}'_i \trianglelefteq_Y \mathcal{M}_i$*. Let* $\mathcal{M} = (\mathcal{F}, \mathcal{P}) = \bullet_{i \in I} \mathcal{M}_i$ *and* $\mathcal{M}' = (\mathcal{F}', \mathcal{P}') = \bullet_{i \in I} \mathcal{M}'_i$*. Then*

(1) each product p of $\mathcal{M}'$ *such that* $p \subseteq Y$ *is a product of* $\mathcal{M}$*, and*
(2) for each set of features $c \subseteq Y$ *and for each product p of* $\mathcal{M}$ *such that* $c \subseteq p$*, there exists a product q of* $\mathcal{M}'$ *such that* $c \subseteq q \subseteq p$.

Proof. (1) Consider a product $p \in \mathcal{P}'$. By construction, for every $i \in I$, there exists $p_i \in \mathcal{P}'_i$ such that $p = \bigcup_{i \in I} p_i$ and, for all $i, j \in I, p_j \cap \mathcal{F}'_i = p_i \cap \mathcal{F}'_i$. By Definition 7, for all $i \in I$, since $p_i \in \mathcal{P}'_i$, we have that $p_i \in \mathcal{P}_i$. Let us now consider $i, j \in I$. We have that $p_i \cap \mathcal{F}_j = p_i \cap Y \cap \mathcal{F}_j = p_i \cap \mathcal{F}'_j = p_j \cap \mathcal{F}'_i = p_j \cap Y \cap \mathcal{F}_i = p_j \cap \mathcal{F}_i$. Hence, $p = \bigcup_{i \in I} p_i \in \mathcal{P}$.

(2) By Definition 7, since $\mathcal{M}'_i \preceq_Y \mathcal{M}_i$, we have $\Pi_Y(\mathcal{M}_i) \preceq \mathcal{M}'_i$. Then, for all $i \in I$, there exists Y_i such that $c \subseteq Y \subseteq Y_i$ and $\Pi_{Y_i}(\mathcal{M}_i) = \mathcal{M}'_i$. Consider a product $p \in \mathcal{P}$ such that $c \subseteq p$. By definition, for all $i \in I$, there exists $p_i \in \mathcal{P}_i$ such that $p = \bigcup_{i \in I} p_i$ and for all $i, j \in I$, we have $p_i \cap \mathcal{F}_j = p_j \cap \mathcal{F}_i$. Let $q = \bigcup_{i \in I}(p_i \cap Y_i)$. Clearly $c \subseteq q \subseteq p$. Moreover, consider $i, j \in I$; since $p_i \cap \mathcal{F}_j = p_j \cap \mathcal{F}_i$ holds, we have: $(p_i \cap Y_i) \cap (\mathcal{F}_j \cap Y_j) = (p_i \cap \mathcal{F}_j) \cap (Y_i \cap Y_j) = (p_j \cap \mathcal{F}_i) \cap (Y_i \cap Y_j) = (p_j \cap Y_j) \cap (\mathcal{F}_i \cap Y_i)$. Hence $q \in \mathcal{P}'$. □

Example 9 (Using the product-discovery criterion with glibc and g-shell FMs). *Consider the packages* glibc *and* g-shell *of Example 5 and the set* $Y = \{$glibc, glibc:v, tzdata$\}$. *It is easy to see that the minimum cut of* $\mathcal{M}_{\text{glibc}}$ *for* Y *is* $\bot^{\trianglelefteq_Y}(\mathcal{M}_{\text{glibc}}) = (Y, 2^Y \setminus Y)$ *because* tzdata *can not be selected when* glibc *and* glibc:v *are selected. Now consider the package* g-shell *instead. The minimum cut of* $\mathcal{M}_{\text{g-shell}}$ *for* Y *is* $\bot^{\trianglelefteq_Y}(\mathcal{M}_{\text{g-shell}}) = (Y, 2^Y)$. *By the definition of feature model composition, we have that* $\bot^{\trianglelefteq_Y}(\mathcal{M}_{\text{glibc}}) \bullet \bot^{\trianglelefteq_Y}(\mathcal{M}_{\text{g-shell}})$ *is the same as* $\bot^{\trianglelefteq_Y}(\mathcal{M}_{\text{glibc}})$.

Now, due to Theorem 3, we can for example derive that the product $\{$glibc, tzdata$\}$ *that contains the shared feature* tzdata *is also a product of the composition of* $\mathcal{M}_{\text{glibc}}$ *and* $\mathcal{M}_{\text{g-shell}}$. *Note that to discover this fact, we avoided computing the composition of the entire feature models and could ignore, e.g., features such as* glibc:doc *and* g-shell.

The criteria provided by Theorem 3 allow us to prove that the lazy product-discovery algorithm (Listing 1 in Section 2) is correct and complete.

Theorem 4 (Soundness and completeness of lazy product discovery). *Given a finite set* I *of indices, a set of feature models* $S = \{\mathcal{M}_i = (\mathcal{F}_i, \mathcal{P}_i) \mid i \in I\}$ *such that all products of* $\mathcal{M}_i$ *are finite, and a finite configuration* c, *the lazy product-discovery algorithm (Listing 1) applied to* S *and* c *always finishes and returns a product of* $\bullet_{i \in I} \mathcal{M}_i$ *that contains* c *if and only if such a product exists.*

Proof. Recall the definitions of auxiliary functions (Section 2):

(1) `pick_cut`$(\mathcal{M}, Y) = \mathcal{M}'$ for some $\mathcal{M}'$ s.t. $\mathcal{M}' \trianglelefteq_Y \mathcal{M}$,
(2) `compose`$(\{\mathcal{M}_1, \ldots, \mathcal{M}_n\}) = \mathcal{M}_1 \bullet \cdots \bullet \mathcal{M}_n$,
(3) `select`$(\mathcal{M}, c)$ is a product of $\mathcal{M}$ containing all the features in c if such a product exists, **None** otherwise;

and the loop invariants **Inv1**–**Inv4** on Line 6. In Section 2 we have already shown that the invariants **Inv1** and **Inv2** hold, and that the algorithm always finishes (because the set of examined features Y, which strictly increases during each traversal of the **while** loop, is bounded by $(\bigcup_{i \in I} \bigcup_{p \in \mathcal{P}_i} p) \cup c$, which is finite by hypothesis). We can now conclude the proof by observing that the invariants **Inv3** and **Inv4** follow straightforwardly from Theorem 3(1) and Theorem 3(2), respectively. □

It is worth observing that a suitable structure of the feature models can enable a particular efficient implementation of the function `pick_cut`$(\mathcal{M}, Y)$. For instance, if the feature-model $\mathcal{M}$ is propositionally represented with a pair of the form $(\mathcal{F}, f \rightarrow \psi)$ (for some set of features $\mathcal{F}$, feature $f \in \mathcal{F}$ and formula ψ) then, whenever $f \notin Y$, `pick_cut`$(\mathcal{M}, Y)$ can return the feature model $(Y', 2^{Y'})$ with $Y' = Y \cap \mathcal{F}$, which corresponds to the pair $(Y', \textbf{true})$ in propositional representation. Therefore, feature models of the form $(\mathcal{F}, f \rightarrow \psi)$ such that $f \notin Y$ can be filtered away before computing the composition `compose`$(\{$`pick_cut`$(\mathcal{M}, Y) | \mathcal{M} \in S\})$ in Lines 4 and 8 of the algorithm.

6 EVALUATION

With lazy product discovery, we aim to efficiently address the product-discovery problem in huge configuration spaces, consisting of hundreds of thousands of features in tens of thousands of feature models. Therefore, we evaluate the performance of the lazy product-discovery algorithm introduced in Section 2. The proposed algorithm loads feature model fragments by need to examine specific features. A feature is *loaded* during a configuration process if it occurs in one of the loaded feature model fragments. In contrast, *standard* product-discovery algorithms (e.g., [41, 43, 59]) load all the feature models before the product-discovery process starts.

We compare the number of loaded features, the time, and the memory needed to solve a product-discovery problem using a lazy and a standard product-discovery algorithm. In detail, we investigate the following research questions:

RQ 1. *How is the number of loaded features affected by the choice of a lazy or a standard product-discovery algorithm?*

RQ 2. *How are the speed and memory consumption of product discovery affected by the choice of a lazy or a standard product-discovery algorithm?*

In industrial practice, product-discovery tools are often optimized for efficiency at the expense of *completeness*. As a consequence, there may be product-discovery problems for which solutions exist but no solution is found by the tool. We compare the lazy product-discovery algorithm to one such state-of-the-art tool by looking at the percentage of cases in which no product is found by the state-of-the-art tool (although products exists), and at the difference in performance for cases when the state-of-the-art product-discovery tool return a correct answer (that is, it either discovers a product or fails when there are no products). For this purpose, we investigate the following research questions:

RQ 3. *How often does a state-of-the-art product-discovery tool fail because of its incompleteness (i.e., the tool does not discover any product, although there is at least one product)?*

RQ 4. *Is lazy product discovery a feasible alternative to state-of-the-art product-discovery tools in terms of execution time and memory consumption?*

6.1 Experimental Design and Subject

To answer these research questions, we performed experiments on an industrial system with a huge configuration space. We chose Gentoo, a source-based Linux distribution with highly-configurable packages [23], which is among the largest fragmented feature models studied in the literature [37]. The experiments were performed on the March 1st 2019 version of the distribution, that contained 36197 feature models with 671617 features overall.

There are no standard benchmarks for product reconfiguration requests. Therefore, we constructed a set of 1000 product-discovery problems for the evaluation. The problems were generated by randomly selected a set of features (between one and ten) such that each of these features requires the installation of a different package. Solving a product-discovery problem c in this context amounts to computing a Gentoo product that includes any version of the packages associated to the features in c and of other packages such that that all dependencies are fulfilled.

We implemented the algorithm of Listing 1 as a tool. This tool, called pdepa, targets Gentoo's package dependencies, which are defined using an ad-hoc syntax [22]. As shown in Example 1, Gentoo's dependencies can be encoded into feature models where features represent both packages and configuration options (called *use flags* in Gentoo). pdepa parses a package dependency and generates the equivalent propositional formula representing the package feature model. A particularity of Gentoo is that the feature model of a package f can be translated into a propositional representation of the form $(\mathcal{F}, f \rightarrow \psi)$, where a package selection feature f represents the package f. The pdepa tool exploits this structure of the feature model in the implementation of the key functions `pick_cut` and `compose` by using the optimization discussed at the end of Section 5. Specifically, pdepa can avoid loading the feature models of packages whose package selection feature is not in the set Y of required features, when composing cuts (Listing 1, Lines 4 and 8).

As its solving engine, pdepa uses the state-of-the-art SMT solver Z3 [19], known for its performance and expressivity. Solvers such as Z3 allow constraints to be added incrementally, reusing part of the search done previously without always restarting the search from scratch. This is extremely useful for composing cuts (Listing 1, Lines 4 and 8) since the existing constraints can be reused, only adding incrementally the new constraints not implied by the existing ones. Although this does not formally reduce the complexity of the algorithm, which is NP-hard in the worst case,[3] in practice these optimizations enable a significant speed-up.

To investigate the research question *RQ 2*, we need to compare pdepa to a standard product-discovery algorithm. Unfortunately, there is no off-the-shelf complete product-discovery tool for Gentoo and therefore we implemented one to establish a baseline for our experiments. We constructed a software that loads all the feature models of all the Gentoo packages and then, as done by pdepa, calls the SMT solver Z3 [19] to solve the configuration problem. We then compared the results of pdepa to the corresponding results of this *baseline tool* (*baseline* for short) in terms of computation time and memory consumption. To ensure a fair comparison, we employ a white-box evaluation, and both pdepa and the baseline use the same implementation for translating the Gentoo dependencies and for loading the feature models.

For research questions *RQ 3* and *RQ 4*, we compare the results of pdepa to the corresponding results of optimized, heuristics-based product-discovery with emerge, the command-line interface to Gentoo's official package manager and distribution system Portage, which is not complete (i.e., it fails to solve some product-discovery problems that have solutions).

All experiments were performed on virtual machines provided by the IaaS OpenStack cloud of the University of Oslo.[4] Every virtual machine had 8 GB of RAM, 2 vCSPUs (2.5 GHz Intel Haswell processors), and was running an Ubuntu 19.04 operating system. The Gentoo operating system was virtualized by running Docker and the image used for the experiments is publicly available.[5]

6.2 Results and Discussion

This section is organized according to research questions RQ1–RQ4. To facilitate the discussion of the experiments, the figures presenting the different results use a fixed ordering of the 1000 product-discovery problems we considered along the x-axis; this ordering is determined by the number of features loaded by pdepa during its computation for a given problem. Each of the 1000 experiments was repeated 5 times for pdepa, for emerge and for the baseline; Figures 1–5 report the mean values for each experiment.

RQ 1. Figure 1 shows the results of the experiments for research question RQ 1 and reports on the number of features loaded by pdepa to solve each product-discovery problem. To highlight how lazy product discovery performs compared to standard product discovery, which needs to load all features before the analysis can start, these numbers are shown as the percentage of features from the full feature model, for each of the product discovery problems. The product-discovery problems have been sorted along the x-axis according to this percentage. The figure shows the loaded features as a full line, the mean number for all the product discovery problems as a dashed line, and the standard deviation (abbreviated to SD in the figures) as a the bar. We see that for the considered product-discovery problems, the mean number of loaded features is only 1.53% of the overall number of features. In summary, the gain in loaded features when solving each of the considered 1000 product-discovery problems using lazy product discovery over standard product discovery is significant.

RQ 2. For research question RQ 2, we compared the speed and memory consumption of product discovery when using pdepa and the baseline on the defined product-discovery problems. For each problem, pdepa loads parts of the FM and calls Z3 incrementally (until a valid product for the whole FM is found), while the baseline first loads the whole FM and then calls Z3.

Figure 2 shows the computation time for product discovery using pdepa (green line) and Figure 3 shows the computation time for product discovery using the baseline. The mean execution time for the baseline is 949 seconds, compared to 78 seconds for pdepa. The minimum and maximum execution times of the baseline are 861.9 and 1222.6 seconds, respectively. The standard deviation for the baseline is negligible (around 35 seconds). It is worth mentioning that about one third of the execution time is devoted to loading the overall feature model, while the remaining time is taken by Z3. The minimum and maximum execution time of pdepa are 1.7 and 155.22 seconds, respectively. The standard deviation is lower than

[3] The NP-hardness derives immediately from the NP-hardness of the problem of finding a valid model for a propositional formula.

[4] https://www.uio.no/english/services/it/hosting/iaas/

[5] https://hub.docker.com/r/gzoumix/pdepa

the one for the baseline, about 18 seconds. The maximum computation time of pdepa is less than one third of the computation time used by the baseline to simply load the overall feature model, and it is about the 16% of the minimum execution time of the baseline.

Figure 4 shows the memory consumption for product discovery using pdepa (green line) and Figure 5 shows the memory consumption for the baseline. The mean memory consumption for the baseline is 3,919.4 MB, compared to 400.715 MB for pdepa. The minimum and maximum memory consumption of the baseline are 3016 and 3980 MB, respectively. About 1 GB of the used memory here is for the feature model itself. The standard deviation for the baseline is negligible (about 70.84 MB). The 7 memory consumption values that fall outside the standard deviation correspond to the product discovery problems that have no solution. The minimum and maximum memory consumption of pdepa are 73 and 620 MB, respectively. The standard deviation, 67.38 MB, is about the same as for the baseline. The maximum memory consumption of pdepa is about 19.62% of the minimum memory consumption of the baseline.

The experiments show a clear correlation between the time and the memory taken by pdepa to solve a product-discovery problem and the number of features loaded by pdepa (cf. Figure 1).

In summary, the experiments clearly demonstrate that lazy product discovery allows significant speed-up and significant reduction of memory consumption, compared to standard product discovery.

RQ 3. We investigated the failures of a heuristics-based incomplete product-discovery tool (emerge) compared to the cases when the complete lazy product discovery algorithm showed that no solution exists, for the 1000 considered product-discovery problems. Figure 6 shows the product-discovery problems for which emerge does not find a product (red and blue bars). For the considered product-discovery problems, emerge fails to find a valid configuration in 26.7% of the cases. In 0,7% of the cases (red bars), no solution exists. Therefore, in 26% of the cases, emerge fails to solve a product-discovery problem that has a solution. The experiments show an interesting correlation between the failures of emerge observed in Figure 6 and the number of features loaded by pdepa during the product-discovery process: the failures of emerge occur more frequently as the number of loaded features needed for lazy product discovery increases. This can be seen since the sorting of the x-axis is the same in Figures 1 and 6. In summary, on 1000 randomly selected product-discovery problems, emerge fails to find a solution that exists in around 26% of the cases.

RQ 4. For research question RQ 4, we investigated how well pdepa performs as an alternative to the state-of-the-art configuration tool emerge. Figure 2 shows the time for product discovery using pdepa (green line) and emerge (blue line). The light green and the light blue bars show the standard deviations and the correspondingly colored dashed lines show the mean times in seconds for pdepa and emerge, respectively. The difference in mean times suggests that pdepa is 11.29 times slower than emerge in average, which corresponds to 70 additional seconds. However, as the results for RQ 3 above shows that emerge fails for a significant number of the considered product-discovery problems, lazy product discovery appears to be a feasible alternative to emerge.

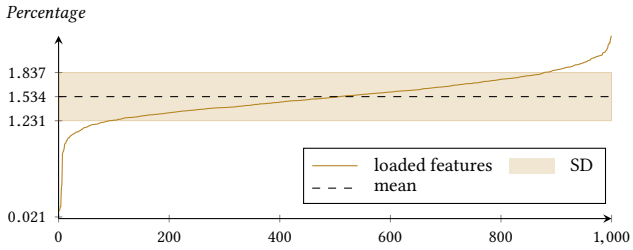

Figure 1: Features loaded by pdepa.

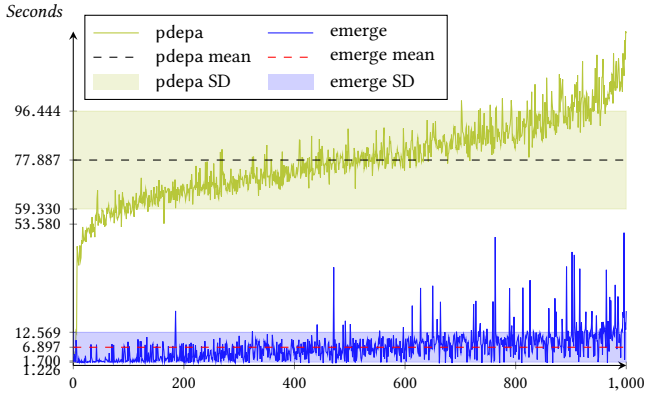

Figure 2: Execution times for pdepa and emerge.

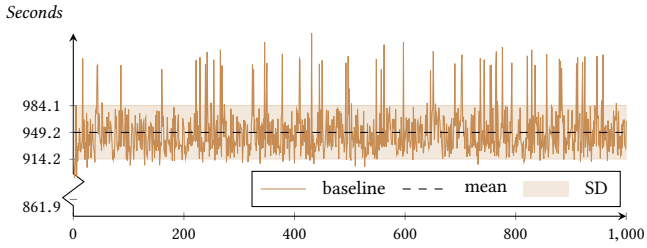

Figure 3: Baseline execution time.

Figure 4 shows the memory consumption for product discovery using pdepa (green line) and emerge (blue line). The light green and the light blue bars show the standard deviations and the corresponding colored dashed lines show the mean memory consumption in MB for pdepa and emerge, respectively. The difference in mean times suggests that pdepa consumes four times more memory than emerge in average (which amounts to around 300 MB).

In summary, lazy product discovery appears as a feasible alternative to emerge if around one order of magnitude additional computation time and four times additional memory consumption are acceptable to always find products when these exist.

6.3 Threats to Validity

6.3.1 External Validity. The results of the evaluation strongly depend on the product-discovery problems considered in the experiments, i.e., on the feature models of the Gentoo packages identified by the features in each product-discovery problem. Due to the lack of standard benchmarks, we considered 1000 product-discovery

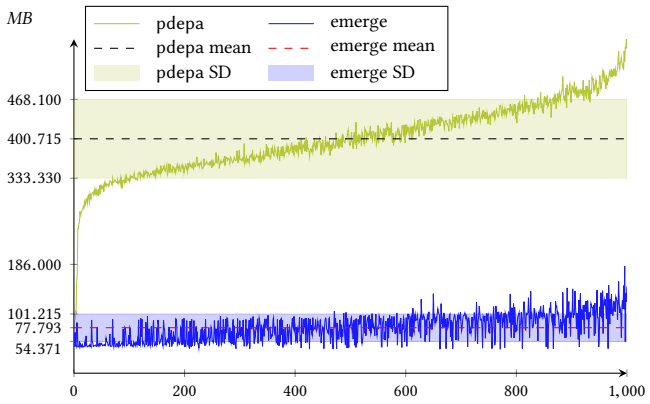

Figure 4: Memory consumption for pdepa and emerge.

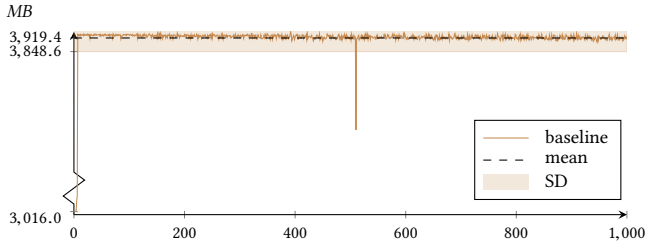

Figure 5: Baseline memory consumption.

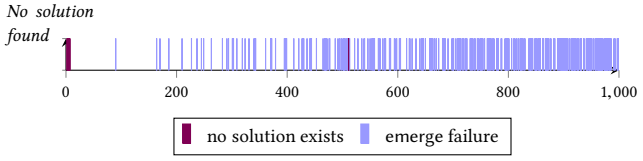

Figure 6: Product-discovery problems with no solution and emerge failures.

problems that were randomly selected from the 671617 features of the March 1st 2019 version of the Gentoo distribution. The random selection used the standard `random` python library [25], that allows to get a set of elements uniformly chosen from a given set.

Different product-discovery problems could potentially lead to different results. We plan to investigate other product-discovery problems for Gentoo and for other domains to get more insights. In particular, it would be interesting to investigate how lazy product discovery performs when varying both the size and the amount of interdependencies of the feature models (see Section 2).

6.3.2 Internal Validity. We used prototype implementations of the lazy product-discovery algorithm and of the standard product-discovery algorithm. Both implementations rely on the Z3 solver [19]. Z3 was chosen because it is a mature solver and freely available. The standard product-discovery algorithm just performs a call to the Z3 solver. The lazy product-discovery algorithm calls the Z3 solver whenever a new feature fragment is loaded. Using a different solver than Z3 may affect the execution time and memory consumption of both the standard and the lazy product-discovery algorithms. We plan to repeat the experiments using another solver.

Introducing optimizations in the lazy product-discovery algorithm could potentially reduce the number of loaded features, the execution time, and the memory consumption for the algorithm. One possible optimization could be to pre-compute at compile time the modal implication graphs [18, 34] of features, which could potentially avoid loading feature models that, e.g., are found to be conflicting in the pre-analysis. Another possible optimization could be the definition and usage of an ad-hoc search strategy for the back-end solver, instead of using solver's default search strategy.

Another threat to validity is that Gentoo's package dependencies are not formally specified, but only given in a textual representation. To reduce the probability of errors in the implementation of the lazy product-discovery algorithm, we have used unit tests to compare the results of pdepa with known correct products. These unit tests were performed by extending the package repository of portage with custom testing and interdependent packages.

Possible bugs in Gentoo's package manager may also be considered a threat to validity. When performing the experiments, we identified the following surprising behavior in emerge:

(1) For some sets of packages[6], emerge implements a heuristic that only considers the feature model of the most recent package in the set, thus forgetting possible solutions.
(2) For emerge to consider a package, some part of its feature model must be configured. Specifically, some of its features must be selected or deselected such that the constraint identified by the variable REQUIRED_USE [22] evaluates to *true*.
(3) For a given product-discovery problem, the dependency analysis of emerge considers each package individually. This can trigger the installation of a package in conflict with the rest of the product-discovery problem, thus preventing the product-discovery problem to be solved even if it has a solution.

We reported these issues to the Gentoo developer community, which replied that they could be considered as bugs of emerge.

We were not able to install the Gentoo variants corresponding to the products discovered by pdepa because of Bug (3) above. Indeed, in many cases, emerge's dependency solver triggers the installation of packages that conflict with pdepa's solution. We plan to overcome this limitation by extending pdepa into a complete package installation tool for Gentoo.

7 RELATED WORK

We discuss related work on interfaces, composition, and configuration of feature models.

Interfaces of Feature Models. The feature-model cut in this paper strengthens the feature-model interfaces introduced by Schröter *et al.* [51], which, as pointed out in Section 3.2, are closely related to *feature model slices* introduced by Acher *et al.* [4]. In the work of Acher *et al.* [4], the focus is on feature model decomposition. In

[6]These sets consisted of packages with an identical SLOT [22]. SLOTs are used in portage to identify which versions of the same package can coexist in one system.

subsequent work [2], Acher *et al.*address evolutionary changes for extracted variability models by using the slice operator in combination with a merge operator, and focus on detecting differences between feature-model versions during evolution. Instead, Schröter *et al.* [51] study how feature model interfaces can be used to support evolution for a feature model composed from feature models fragments. Changes to fragments which do not affect their interfaces do not require the overall feature model to be rebuilt (by composing the fragments) in order to reanalyze it. Challenges encountered to support evolution in software product line engineering have previously been studied by Dhungana *et al.* [20]. They use interfaces to hide information in feature model fragments and save a merge history of fragments to give feedback and facilitate fragment maintenance. No automated analysis is considered. In contrast to this work on feature model interfaces for evolution, the cut in our work is for efficient automated product discovery in huge feature models represented as interdependent feature model fragments.

Feature-model views [30, 39, 50] focus on a subset of the relevant features of a given feature model, similarly to feature-model interfaces. Different views regarding one master feature model are used to capture the needs of different stakeholders, so that a product of the master feature model can be identified based on the views' partial configurations. This work on multiple views to a product in a feature model is orthogonal to our work on feature-model cuts, which targets the efficient configuration of systems comprising many interdependent configurable packages.

Composition of Feature Models. Feature-model composition is often used for multi software product lines (i.e., sets of interdependent product lines) [29, 35, 37, 47]. Eichelberger and Schmid [21] provide an overview of textual-modeling languages which support variability-model composition (like FAMILIAR [5], VELVET [49], TVL [16], VSL [1]) and compare how they support composition, modularity, and evolution. Acher *et al.* [6] compare different feature-model composition operators by considering possible implementations and discuss advantages and drawbacks. For the investigation of efficient automated configuration of huge feature models in this paper, we use the propositional representation of feature models and a composition operator that corresponds to logical conjunction.

Configuration of Feature Models. Product discovery (also called product configuration or product derivation) is the process of selecting and deselecting features in a feature model in order to obtain a product [26]. This is a central and widely studied problem in the field of automated reasoning [9]; e.g., more than 50 different methods for product discovery are discussed in a recent survey [26].

We are not aware of any method that addresses how complete and efficient product-discovery can be achieved in configuration spaces comprising different interdependent feature model fragments without composing all the fragments. The tool for lazy product discovery is in the class of product discovery tools which *automatically* produce valid configurations.

Automated configuration is supported by a number of tools, including FeatureIDE [59], GEARS [36], GUIDSL [8], IBED [60], HyVarRec [40], SATIBEA [27] S2T2 Configurator [15], SIP [28], SPL Conqueror [54], S.P.L.O.T. [43], and VariaMos [42]. However, in contrast to our work, all these tools are eager and require the building of the global feature model by composing all its fragments. As such, these tools are in line with the standard product discovery algorithm, as discussed in Section 6.

Some of these standard product discovery tools are interactive, i.e., they support and interact with the user by guiding her in producing a valid configuration or finding one that maximizes her preferences [8, 15, 42, 43]. Our method for lazy product discovery can be exploited to support interactive product discovery either (i) by requiring the user to enter preferences over different configurations or (ii) by interacting with the user when deciding what partial configuration should be extended (i.e., when the `select` function of the algorithm in Listing 1 is performed). An extension of the lazy product discovery algorithm in this direction is left as future work.

Different computational techniques can be used to solve the product discovery problem: satisfiability solvers, constraint programming, evolutionary algorithms, stochastic algorithms, or binary decision diagrams [9, 10, 46]. Due to the NP-hardness of the configuration problem itself, most complete approaches rely on SAT solvers [31, 44], but more recently, the use of more powerful backend solvers, such as constraint solvers and SMT solvers, are starting to be explored for automatic configuration of feature models [11, 41, 45, 57]. In our work, we have used Z3 [19] which is one of the most powerful and mature SMT solvers available today. We would like to remark, however, that the lazy product discovery method itself is orthogonal to the tool chosen, as long as the backend solver allows to implement the `pick_cut`, `compose`, and `select` operations of Listing 1.

8 CONCLUSION AND FUTURE WORK

Product discovery in huge configuration spaces represented as sets of interdependent feature models is challenging. Standard analysis techniques for fragmented feature models require all the feature models to be composed in order to apply the analysis. Recent work has shown that several analyses of fragmented feature models can be simplified using techniques such as feature model interfaces and slicing, however these techniques do not work for product discovery in sets of interdependent feature models.

In this paper, we introduce a method for automated product discovery in configuration spaces represented as sets of interdependent feature models. The method is lazy as features are added incrementally to the analysis until a product is found. We introduce and formalize the feature model cut, and leverage this concept to define a product-discovery criterion. We exploit this criterion to define a complete and efficient algorithm for lazy product discovery in sets of interdependent feature models. We have evaluated the potential of lazy product discovery on randomly constructed configuration problems for the configuration space of the source-based Linux distribution Gentoo, with 36197 interdependent feature models and a total of 671617 features. The evaluation has demonstrated significant gains compared to standard product discovery and that the trade-off of performance for completeness is reasonable compared to the heuristics-based product-discovery with emerge, the

command-line interface to Gentoo's official package manager and distribution system Portage.

We are now investigating different optimizations of the current prototype, such as the exploitation of modal implication graphs pre-computed at compile time and the usage of ad-hoc SMT search strategies. In future work we plan to investigate other product-discovery problems for Gentoo as well as for other domains, to gain more insights into lazy product discovery. While our results make us confident that lazy product discovery is a viable method for product discovery in huge configuration spaces, we believe that it may also be used to complement optimized but incomplete algorithms when these fail, such as emerge for Gentoo. We also plan to investigate how lazy product discovery can be combined with interactive product discovery.

ACKNOWLEDGMENTS

This work is partially funded by the Sirius Center for Scalable Data Access and the Compagnia di San Paolo. We thank the reviewers for constructive feedback, Thomas Thüm, Andrzej Wasowski and Sven Apel for useful discussions on the topic of this paper, and Simone Donetti for testing the publicly available artifact.

REFERENCES

[1] Andreas Abele, Yiannis Papadopoulos, David Servat, Martin Törngren, and Matthias Weber. 2010. The CVM Framework - A Prototype Tool for Compositional Variability Management. In *Proc. 4th International Workshop on Variability Modelling of Software-Intensive Systems (VaMoS 2010) (ICB-Research Report)*, Vol. 37. Universität Duisburg-Essen, 101–105. http://www.vamos-workshop.net/proceedings/VaMoS_2010_Proceedings.pdf

[2] Mathieu Acher, Anthony Cleve, Philippe Collet, Philippe Merle, Laurence Duchien, and Philippe Lahire. 2014. Extraction and Evolution of Architectural Variability Models in Plugin-based Systems. *Software and Systems Modeling* 13, 4 (Oct. 2014), 1367–1394. https://doi.org/10.1007/s10270-013-0364-2

[3] Mathieu Acher, Philippe Collet, Philippe Lahire, and Robert B. France. 2010. Comparing Approaches to Implement Feature Model Composition. In *Proc. 6th European Conference on Modelling Foundations and Applications (ECMFA 2010)*, Thomas Kühne, Bran Selic, Marie-Pierre Gervais, and François Terrier (Eds.). Springer, 3–19.

[4] Mathieu Acher, Philippe Collet, Philippe Lahire, and Robert B. France. 2011. Slicing feature models. In *Proc. 26th International Conference on Automated Software Engineering (ASE 2011)*. IEEE Computer Society Press, 424–427. https://doi.org/10.1109/ASE.2011.6100089

[5] Mathieu Acher, Philippe Collet, Philippe Lahire, and Robert B. France. 2013. FAMILIAR: A domain-specific language for large scale management of feature models. *Science of Computer Programming* 78, 6 (2013), 657–681. https://doi.org/10.1016/j.scico.2012.12.004

[6] Mathieu Acher, Benoît Combemale, Philippe Collet, Olivier Barais, Philippe Lahire, and Robert B. France. 2013. Composing Your Compositions of Variability Models. In *Proc. 16th International Conference on Model-Driven Engineering Languages and Systems (MODELS 2013)*, Ana Moreira, Bernhard Schätz, Jeff Gray, Antonio Vallecillo, and Peter J. Clarke (Eds.). Springer, 352–369. https://doi.org/10.1007/978-3-642-41533-3_22

[7] Sven Apel, Don S. Batory, Christian Kästner, and Gunter Saake. 2013. *Feature-Oriented Software Product Lines: Concepts and Implementation.* Springer.

[8] Don Batory. 2005. Feature Models, Grammars, and Propositional Formulas. In *Proc. 9th International Software Product Line Conference (SPLC 2005)*. Springer, 7–20.

[9] David Benavides, Sergio Segura, and Antonio Ruiz-Cortés. 2010. Automated analysis of feature models 20 years later: A literature review. *Information Systems* 35, 6 (2010), 615–636. https://doi.org/10.1016/j.is.2010.01.001

[10] David Benavides, Sergio Segura, Pablo Trinidad, and Antonio Ruiz-Cortés. 2007. FAMA: Tooling a framework for the automated analysis of feature models. In *Proc. 1st International Workshop on Variability Modelling of Software-Intensive Systems (VaMoS 2007) (Lero Technical Report)*, Vol. 2007-01. 129–134.

[11] David Benavides, Pablo Trinidad, and Antonio Ruiz Cortés. 2005. Using Constraint Programming to Reason on Feature Models. In *Proc. 17th International Conference on Software Engineering and Knowledge Engineering (SEKE 2005)*. 677–682. http://ksiresearchorg.ipage.com/seke/Proceedings/seke/SEKE2005_Proceedings.pdf

[12] Thorsten Berger, Ralf Rublack, Divya Nair, Joanne M. Atlee, Martin Becker, Krzysztof Czarnecki, and Andrzej Wąsowski. 2013. A survey of variability modeling in industrial practice. In *Proc. 7th International Workshop on Variability Modelling of Software-Intensive Systems (VaMoS 2013)*, Stefania Gnesi, Philippe Collet, and Klaus Schmid (Eds.). ACM Press, 7:1–7:8.

[13] Thorsten Berger, Steven She, Rafael Lotufo, Andrzej Wąsowski, and Krzysztof Czarnecki. 2010. Variability modeling in the real: a perspective from the operating systems domain. In *Proc. 25th International Conference on Automated Software Engineering (ASE 2010)*, Charles Pecheur, Jamie Andrews, and Elisabetta Di Nitto (Eds.). ACM Press, 73–82.

[14] Marko Bošković, Gunter Mussbacher, Ebrahim Bagheri, Daniel Amyot, Dragan Gašević, and Marek Hatala. 2010. Aspect-Oriented Feature Models. In *Proc. Models in Software Engineering - Workshops and Symposia at MODELS 2010*, Jürgen Dingel and Arnor Solberg (Eds.). Springer, 110–124.

[15] Goetz Botterweck, Mikolás Janota, and Denny Schneeweiss. 2009. A Design of a Configurable Feature Model Configurator. In *Proc. 3rd International Workshop on Variability Modelling of Software-Intensive Systems (VaMoS 2009) (ICB Research Report)*, Vol. 29. Universität Duisburg-Essen, 165–168. http://www.vamos-workshop.net/proceedings/VaMoS_2009_Proceedings.pdf

[16] Andreas Classen, Quentin Boucher, and Patrick Heymans. 2011. A text-based approach to feature modelling: Syntax and semantics of TVL. *Science of Computer Programming* 76, 12 (2011), 1130 – 1143. https://doi.org/10.1016/j.scico.2010.10.005

[17] E. F. Codd. 1970. A Relational Model of Data for Large Shared Data Banks. *Commun. ACM* 13, 6 (1970), 377–387. https://doi.org/10.1145/362384.362685

[18] Roberto Di Cosmo and Jérôme Vouillon. 2011. On software component co-installability. In *Proc. 19th Symposium on the Foundations of Software Engineering (FSE-19) and 13th European Software Engineering Conference (ESEC-13)*. ACM Press, 256–266. https://doi.org/10.1145/2025113.2025149

[19] Leonardo Mendonça de Moura and Nikolaj Bjørner. 2008. Z3: An Efficient SMT Solver. In *Proc. 14th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS 2008)*, C. R. Ramakrishnan and Jakob Rehof (Eds.). Springer, 337–340.

[20] Deepak Dhungana, Paul Grünbacher, Rick Rabiser, and Thomas Neumayer. 2010. Structuring the modeling space and supporting evolution in software product line engineering. *Journal of Systems and Software* 83, 7 (2010), 1108 – 1122. https://doi.org/10.1016/j.jss.2010.02.018

[21] Holger Eichelberger and Klaus Schmid. 2013. A Systematic Analysis of Textual Variability Modeling Languages. In *Proc. 17th International Software Product Line Conference (SPLC 2013)*. ACM Press, 12–21. https://doi.org/10.1145/2491627.2491652

[22] Gentoo Foundation. 2017. Package Manager Specification. (2017). https://dev.gentoo.org/~ulm/pms/head/pms.html Last visited, 2019-08-20.

[23] Gentoo Foundation. 2019. Gentoo Linux. (2019). https://gentoo.org Last visited, 2019-08-20.

[24] Gentoo Foundation. 2019. Portage - Gentoo Wiki. (2019). https://wiki.gentoo.org/wiki/Portage Last visited, 2019-08-20.

[25] Python Software Foundation. 2019. random — Generate pseudo-random numbers. (2019). https://docs.python.org/3/library/random.html Last visited, 2019-08-20.

[26] José A. Galindo, David Benavides, Pablo Trinidad, Antonio Manuel Gutiérrez-Fernández, and Antonio Ruiz-Cortés. 2019. Automated analysis of feature models: Quo vadis? *Computing* 101, 5 (2019), 387–433. https://doi.org/10.1007/s00607-018-0646-1

[27] Christopher Henard, Mike Papadakis, Mark Harman, and Yves Le Traon. 2015. Combining Multi-Objective Search and Constraint Solving for Configuring Large Software Product Lines. In *Proc. 37th International Conference on Software Engineering (ICSE 2015)*. IEEE Computer Society Press, 517–528. https://doi.org/10.1109/ICSE.2015.69

[28] Robert M. Hierons, Miqing Li, Xiaohui Liu, Sergio Segura, and Wei Zheng. 2016. SIP: Optimal Product Selection from Feature Models Using Many-Objective Evolutionary Optimization. *ACM Transactions on Software Engineering and Methodology* 25, 2 (2016), 17:1–17:39. https://doi.org/10.1145/2897760

[29] Gerald Holl, Paul Grünbacher, and Rick Rabiser. 2012. A systematic review and an expert survey on capabilities supporting multi product lines. *Information & Software Technology* 54, 8 (2012), 828–852. https://doi.org/10.1016/j.infsof.2012.02.002

[30] Arnaud Hubaux, Patrick Heymans, Pierre-Yves Schobbens, and Dirk Deridder. 2010. Towards Multi-view Feature-Based Configuration. In *Proc. 16th International Working Conference on Requirements Engineering: Foundation for Software Quality (REFSQ 2010)*, Roel J. Wieringa and Anne Persson (Eds.). Springer, 106–112.

[31] Mikolás Janota. 2008. Do SAT Solvers Make Good Configurators?. In *Proc. 12th International Software Product Line Conference (SPLC 2008) Workshops*. Lero Int. Science Centre, University of Limerick, Ireland, 191–195.

[32] Kyo Kang, Sholom Cohen, James Hess, William Novak, and A. Peterson. 1990. *Feature-Oriented Domain Analysis (FODA) Feasibility Study.* Technical Report

CMU/SEI-90-TR-021. Software Engineering Institute, Carnegie Mellon University. http://resources.sei.cmu.edu/library/asset-view.cfm?AssetID=11231

[33] S. C. Kleene. 1938. On notation for ordinal numbers. *Journal of Symbolic Logic* 3, 4 (1938), 150–155. https://doi.org/10.2307/2267778

[34] Sebastian Krieter, Thomas Thüm, Sandro Schulze, Reimar Schröter, and Gunter Saake. 2018. Propagating Configuration Decisions with Modal Implication Graphs. In *Proc. 40th International Conference on Software Engineering (ICSE 2018)*. ACM Press, 898–909. https://doi.org/10.1145/3180155.3180159

[35] Charles W. Krueger. 2006. New Methods in Software Product Line Development. In *Proc. 10th International Software Product Line Conference (SPLC 2006)*. IEEE Computer Society Press, 95–102. https://doi.org/10.1109/SPLINE.2006.1691581

[36] Charles W. Krueger and Paul Clements. 2018. Feature-based systems and software product line engineering with gears from BigLever. In *Proc. 22nd International Software Product Line Conference (SPLC 2018)*. ACM Press, 1–4. https://doi.org/10.1145/3236405.3236409

[37] Michael Lienhardt, Ferruccio Damiani, Simone Donetti, and Luca Paolini. 2018. Multi Software Product Lines in the Wild. In *Proc. 12th International Workshop on Variability Modelling of Software-Intensive Systems (VaMoS 2018)*. ACM Press, 89–96. https://doi.org/10.1145/3168365.3170425

[38] Rafael Lotufo, Steven She, Thorsten Berger, Krzysztof Czarnecki, and Andrzej Wąsowski. 2010. Evolution of the Linux Kernel Variability Model. In *Proc. 14th International Software Product Line Conference (SPLC 2010)*, Jan Bosch and Jaejoon Lee (Eds.). Springer, 136–150. https://doi.org/10.1007/978-3-642-15579-6_10

[39] Mike Mannion, Juha Savolainen, and Timo Asikainen. 2009. Viewpoint-Oriented Variability Modeling. In *Proc. 33rd International Computer Software and Applications Conference (COMPSAC 2009)*. IEEE Computer Society Press, 67–72. https://doi.org/10.1109/COMPSAC.2009.19

[40] Jacopo Mauro, Michael Nieke, Christoph Seidl, and Ingrid Chieh Yu. 2016. Context Aware Reconfiguration in Software Product Lines. In *Proc. 10th International Workshop on Variability Modelling of Software-intensive Systems (VaMoS 2016)*. ACM Press, 41–48. https://doi.org/10.1145/2866614.2866620

[41] Jacopo Mauro, Michael Nieke, Christoph Seidl, and Ingrid Chieh Yu. 2018. Context-aware reconfiguration in evolving software product lines. *Science of Computer Programming* 163 (2018), 139–159. https://doi.org/10.1016/j.scico.2018.05.002

[42] Raúl Mazo, Camille Salinesi, and Daniel Diaz. 2012. VariaMos: a Tool for Product Line Driven Systems Engineering with a Constraint Based Approach. In *Proc. CAiSE'12 Forum at the 24th International Conference on Advanced Information Systems Engineering (CAiSE 2012) (CEUR Workshop Proceedings)*, Vol. 855. CEUR-WS.org, 147–154.

[43] Marcílio Mendonça, Moises Branco, and Donald D. Cowan. 2009. S.P.L.O.T.: software product lines online tools. In *Companion to the 24th Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA 2009)*. ACM Press, 761–762. https://doi.org/10.1145/1639950.1640002

[44] Marcílio Mendonça, Andrzej Wąsowski, and Krzysztof Czarnecki. 2009. SAT-based Analysis of Feature Models is Easy. In *Proceedings of the 13th International Software Product Line Conference (ACM International Conference Proceeding Series)*, Dirk Muthig and John D. McGregor (Eds.), Vol. 446. ACM Press, 231–240.

[45] Raphaël Michel, Arnaud Hubaux, Vijay Ganesh, and Patrick Heymans. 2012. An SMT-based approach to automated configuration. In *Proc. 10th International Workshop on Satisfiability Modulo Theories (SMT 2012) (EPiC Series in Computing)*, Vol. 20. EasyChair, 109–119.

[46] Lina Ochoa, Juliana Alves Pereira, Oscar González Rojas, Harold E. Castro, and Gunter Saake. 2017. A survey on scalability and performance concerns in extended product lines configuration. In *Proc. 11th International Workshop on Variability Modelling of Software-intensive Systems (VaMoS 2017)*. ACM Press, 5–12. https://doi.org/10.1145/3023956.3023959

[47] Marko Rosenmüller and Norbert Siegmund. 2010. Automating the Configuration of Multi Software Product Lines. In *Proc. 4th International Workshop on Variability Modelling of Software-Intensive Systems (VaMoS 2010) (ICB-Research Report)*, Vol. 37. Universität Duisburg-Essen, 123–130.

[48] Marko Rosenmüller, Norbert Siegmund, Christian Kästner, and Syed Saif Ur Rahman. 2008. Modeling Dependent Software Product Lines. In *Proc. Workshop on Modularization, Composition and Generative Techniques for Product Line Engineering (McGPLE)*. Department of Informatics and Mathematics, University of Passau, 13–18.

[49] Marko Rosenmüller, Norbert Siegmund, Thomas Thüm, and Gunter Saake. 2011. Multi-dimensional Variability Modeling. In *Proc. 5th International Workshop on Variability Modelling of Software-Intensive Systems (VaMoS 2011)*. ACM Press, 11–20. https://doi.org/10.1145/1944892.1944894

[50] Julia Schroeter, Malte Lochau, and Tim Winkelmann. 2012. Multi-perspectives on Feature Models. In *Proc. 15th International Conference on Model Driven Engineering Languages and Systems (MODELS 2012)*, Robert B. France, Jürgen Kazmeier, Ruth Breu, and Colin Atkinson (Eds.). Springer, 252–268.

[51] Reimar Schröter, Sebastian Krieter, Thomas Thüm, Fabian Benduhn, and Gunter Saake. 2016. Feature-Model Interfaces: The Highway to Compositional Analyses of Highly-Configurable Systems. In *Proc. 38th International Conference on Software Engineering (ICSE 2016)*. ACM Press, 667–678. https://doi.org/10.1145/2884781.2884823

[52] Reimar Schröter, Thomas Thüm, Norbert Siegmund, and Gunter Saake. 2013. Automated Analysis of Dependent Feature Models. In *Proc. 7th International Workshop on Variability Modelling of Software-Intensive Systems (VaMoS 2013)*, Stefania Gnesi, Philippe Collet, and Klaus Schmid (Eds.). ACM Press, 9:1–9:5. https://doi.org/10.1145/2430502.2430515

[53] Sergio Segura, José A. Galindo, David Benavides, José A. Parejo, and Antonio Ruiz-Cortés. 2012. BeTTy: Benchmarking and Testing on the Automated Analysis of Feature Models. In *Proc. 6th International Workshop on Variability Modelling of Software-Intensive Systems (VaMoS 2012)*. ACM Press, 63–71. https://doi.org/10.1145/2110147.2110155

[54] Norbert Siegmund, Marko Rosenmüller, Martin Kuhlemann, Christian Kästner, Sven Apel, and Gunter Saake. 2012. SPL Conqueror: Toward optimization of non-functional properties in software product lines. *Software Quality Journal* 20, 3-4 (2012), 487–517. https://doi.org/10.1007/s11219-011-9152-9

[55] Reinhard Tartler, Daniel Lohmann, Christian Dietrich, Christoph Egger, and Julio Sincero. 2011. Configuration coverage in the analysis of large-scale system software. *Operating Systems Review* 45, 3 (2011), 10–14.

[56] Reinhard Tartler, Daniel Lohmann, Julio Sincero, and Wolfgang Schröder-Preikschat. 2011. Feature consistency in compile-time-configurable system software: facing the linux 10, 000 feature problem. In *Proc. 6th European Conference on Computer systems (EuroSys 2011)*, Christoph M. Kirsch and Gernot Heiser (Eds.). ACM Press, 47–60.

[57] Thomas Thüm. 2018. (2018). https://github.com/FeatureIDE/FeatureIDE/issues/836

[58] Thomas Thüm, Sven Apel, Christian Kästner, Ina Schaefer, and Gunter Saake. 2014. A Classification and Survey of Analysis Strategies for Software Product Lines. *ACM Comput. Surv.* 47, 1 (2014), 1–45.

[59] Thomas Thüm, Christian Kästner, Fabian Benduhn, Jens Meinicke, Gunter Saake, and Thomas Leich. 2014. FeatureIDE: An extensible framework for feature-oriented software development. *Science of Computer Programming* 79 (2014), 70–85. https://doi.org/10.1016/j.scico.2012.06.002

[60] Yinxing Xue, Jinghui Zhong, Tian Huat Tan, Yang Liu, Wentong Cai, Manman Chen, and Jun Sun. 2016. IBED: Combining IBEA and DE for optimal feature selection in software product line engineering. *Applied Soft Computing* 49 (2016), 1215–1231. https://doi.org/10.1016/j.asoc.2016.07.040