The Journal of Systems and Software 210 (2024) 111938

Contents lists available at ScienceDirect

The Journal of Systems & Software

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

Software product lines and features from the perspective of set theory with an application to feature location☆

Ulrich Eisenecker [a], Richard Müller [b,*]

[a] Information Systems Institute, Leipzig University, Grimmaische Str. 12, 04109 Leipzig, Germany
[b] Deloitte Services GmbH, Lutherstr. 51, 02826, , Görlitz, Germany

ARTICLE INFO

Dataset link: https://github.com/softvis-research/features

Keywords:
Software product line
Variability modeling
Feature model
Feature isolation
Feature location

ABSTRACT

Features are a central concept of Software Product Lines (SPLs). Over the last decades, several understandings of what features are have evolved. They have important similarities but also some differences. However, there is no unifying formal theory of features. We present a definition of features that is completely independent of the representation of features in software artifacts and a comprehensive categorization scheme for features. Based on this, we introduce a formal approach for feature-specific composition and decomposition of SPLs using set theory. We apply this approach to feature isolation as a prerequisite for feature location and provide a theoretical validation. For this purpose, we present programs that compose all possible systems for a given number of features and isolate each feature. By testing various conclusions based on this theory, we evaluate its soundness, consistency, and robustness. The results show under which conditions features can be successfully isolated and possibly located. In addition, we describe the current limitations of the approach and provide an outlook for future work.

1. Introduction

In Software Product Line Engineering (SPLE), variability modeling aims at capturing the commonality and variability among the software-intensive systems of a Software Product Line (SPL) (Pohl and Metzger, 2018). This core activity improves the software development process through reuse and supports developers to increase efficiency, reduce costs, and improve the quality of the systems. In a tertiary study (Raatikainen et al., 2019), the authors identify three categories for dealing with variability in SPLs, namely feature models (FM), orthogonal variability models (OVM), and decision models (DM). This paper focuses on FM, the most well-known and widely used approach for variability modeling.

In FM, the conceptual building blocks are so-called features. Ideally, they represent parts of software artifacts that have exactly the right size as building blocks, that is, they are no more finely granular than necessary and do not need to be tailored further for modeling variability in a SPL as, for example, suggested by Ziadi et al. (2012). Actually, the implementation of features can differ enormously with respect to granularity as pointed out by Kästner et al. (2008) and Michelon et al. (2021). Knowing these features and having their corresponding parts in software artifacts available allows a more systematic and in some application scenarios automated creation of the systems of a SPL. Here and throughout the rest of the paper, we will use the term *system* as a synonym for *product* or *variant*. Using this term is consistent with others papers, for example Ziadi et al. (2012). On the other hand, if systems of a SPL are already available, and it is known or it can be guessed which features each system has, this knowledge can be applied to isolate the single features to eventually locate the parts that implement them (Ziadi et al., 2012). The results of feature location can advance the understanding of SPLs and support reasoning about, for example, estimating functional and non-functional properties of features and systems, creating new systems, and improving testing for relevance and performance (Thüm et al., 2011).

Apel et al. (2008) propose a more formal definition of a feature that is part of an algebra for feature composition. It has to be emphasized that features do not only correspond to code. A feature can be also mapped to parts of any other textual, graphical, or whatever artifacts that belong to a system of a SPL. Bigliardi et al. (2014) provide an overview of non-code software artifacts. In fact, a feature can be mapped to code and non-code artifacts simultaneously. Moreover, a feature does not have to be located in a single part of a software artifact (Kästner et al., 2008; Thüm et al., 2011). It can be also spread over several positions in a software artifact or a group of related software artifacts which can also differ in their nature, that is, code,

☆ Editor: Laurence Duchien.

* Corresponding author.
E-mail addresses: eisenecker@wifa.uni-leipzig.de (U. Eisenecker), richmueller@deloitte.de (R. Müller).

https://doi.org/10.1016/j.jss.2023.111938
Received 13 February 2023; Received in revised form 11 November 2023; Accepted 19 December 2023
Available online 3 January 2024

documentation, and so on. Overall, there are many definitions for the term feature, some of which overlap but also have significant differences. Some of the definitions equate features with parts of software artifacts, others relate features and parts of software artifacts. In this paper, we present a formal definition of the term feature that allows us to reason about features and systems of a SPL independently of the software artifacts they represent.

In FMs, features are usually represented in a tree-like structure, a so-called feature diagram, that shows a root feature, parent and child features, as well as mandatory or optional features and features that are organized in or- or xor-feature groups. The parent-sub-feature relationship usually reflects the dominant stakeholder perspective. Sometimes, not all combinations of the features and their relationships are valid. In these cases, additional requires- and excludes-relationships can be specified for features, mostly in form of textual annotations or a separate documentation. Even more extensions have been proposed for feature diagrams, for example feature attributes, cardinalities for solitary features as well as feature groups, and special feature types like abstract features and aspectual features (Schobbens et al., 2006; Thüm et al., 2011; Zaatar and Hamza, 2011; Wahyudianto et al., 2014; Seidl et al., 2016). Feature models, including feature diagrams, can be used as additional artifact that complements the usual software artifacts with the documentation of common and variable features and dependencies among variable features. They can also be turned in active software artifacts which check or control the configuration of systems of a SPL. Some of the feature models and notations even provide new feature categories and give additional meaning to features through their relationships. This paper presents basic relationships between features that result in a minimum number of categories for features to be distinguished and are completely independent of the software artifacts to which they refer.

There are several tools available, including commercial ones, some of which also support product derivation and/or product generation based on feature models (see El Dammagh and De Troyer (2011) for a historic overview, commercial tools are for example BigLever Software Gears[1] and pure::variants by pure systems[2]).

Besides feature location further questions are if features share code or so-called feature interactions occur (Apel et al., 2013; Soares et al., 2018; Kolesnikov, 2019). If a system implements two features their implementations can have common code parts, and it can happen that dealing with the presence of both features in a system requires to add extra code, which is known as feature interaction. This paper proposes separate categories for features that have common parts in software artifacts and for feature interactions.

Usually, not all systems of a SPL are available in practice. The importance of the available systems of a SPL for the location of features has been pointed out several times, for example by Michelon et al. (2021). The formal approach we present here can be used to determine exactly which features can in principle exist and be located for a given number of systems of a SPL. A follow-up question is: What are the consequences when a new system emerges? Since all systems of a SPL must differ by their features, the advent of a new system can have significant impact on which features can be located and how.

The theory presented in this paper is designed to be unaware of the way how features are represented as artifacts. It completely abstracts from the nature of the software artifacts and provides a unifying view of features on the same level of abstraction. As a consequence, feature categories, the composition of systems by combining features, and feature isolation as a prerequisite of feature location can be described entirely on an abstract level, independent from the mapping of a feature to its concrete representation in software artifacts. With the knowledge of which systems are available to a SPL, it can be determined precisely whether a certain feature can be isolated and thus located at all and how this can be done efficiently. Based on an assumed bundle of feature categories, it can be checked whether certain features are actually present in the software artifacts.

In our initial approach (Müller and Eisenecker, 2019), we briefly sketched a formal approach for feature-specific composition and decomposition of SPLs and implemented a partial solution for the ArgoUML feature location challenge (Martinez et al., 2018a). We were able to locate exclusive traces and pair-wise feature interaction traces to complete classes and methods of 12 out of 24 features and feature combinations with a precision of 1.0 and a recall of 1.0. However, these results were limited to complete class and method traces in the traditional scenario. The remaining 12 features, feature combinations, and feature negations had a precision and recall of 0.0 because they only contain traces of class and method refinements that were not available in the software graph at this time. The latest version of the challenge solution locates all traces of the 24 features with a precision of 1.0 and a recall of 1.0 in the scenario with all configurations.[3]

Here, we will develop the theory further and abstract from the concrete implementation of the artifacts. In summary, the main contributions are as follows.

1. A comprehensive categorization scheme for features including their relationships with the systems of a SPL.
2. A formal approach for feature-specific composition and decomposition of SPLs using set theory.
3. A theoretical validation with a series of programs that generate all systems of a SPL for a given number of features and derive calculation formulas for isolating and eventually locating features.

All programs mentioned in this paper are published on GitHub and are available under the Apache License 2.0.[4]

The remaining part of the paper is structured as follows. In Section 2, we introduce a formal definition of feature categories. Based on this categorization, we develop a theory for feature-specific composition and decomposition of SPLs using set theory. In Section 3, we theoretically validate the approach and evaluate its soundness, consistency, and robustness. Therefore, we present a set of programs that generate all systems of a SPL for a given number of features and derive set differences for isolating features. In Section 4, we outline different directions of future work. In Section 5, we describe current limitations. In Section 6, we discuss related work regarding variability modeling with focus on FM. Furthermore, we examine feature location approaches and compare them with our approach. Finally, we summarize our findings in Section 7.

2. Theory

In the following, we provide definitions of the terms *software artifact* and *feature* and show how they are related in this work. Then, we systematically present the feature categories of this approach. The basis are independent features, from which all dependent features such as and-, or-, and not-features emerge. In addition, excludes- and requires-relationships can be defined for independent features, making them explicitly dependent features. We use propositional logic to represent these relationships. Once these feature categories are available, they can be used to systematically create system descriptions in the form of sets containing the features defining the system in question. In this context, it is useful to distinguish several semantically valid bundles of feature categories. Finally, we show how each feature can be isolated using the previously generated system descriptions.

[1] https://biglever.com/solution/gears/.
[2] https://www.pure-systems.com/de/purevariants.
[3] https://github.com/softvis-research/argouml-spl-benchmark/releases/tag/v2.0.
[4] https://github.com/softvis-research/features.

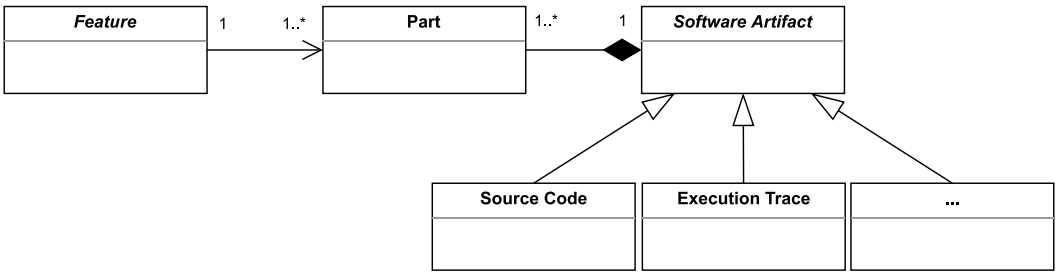


Fig. 1. Relation between feature and software artifact.

2.1. Software artifact

The software product life cycle (SPLC) describes the entire life cycle of a software system from the initial idea through deployment to retirement. It includes both, the phases of the software development life cycle (SDLC) analysis, design, development, and test as well as all other phases to retirement such as deployment, maintenance, and management IEEE Computer Society (2014, p. 8–4). A *software artifact* is an item that can be produced during the entire SPLC. This may be a data model, a design document, source code, an execution trace, or a setup script (see Bigliardi et al. (2014) for more examples of non-code artifacts). Each software artifact can possess specific parts. For example, parts of source code may be type declarations, function definitions, or statements. A part of source code may be concentrated in one place or spread around in many positions. From an abstract point of view, parts of other software artifacts have analogous properties.

2.2. Feature

There are several definitions of a feature in the literature, for example Czarnecki et al. (2004, 2005), Apel et al. (2008) and Rosenmüller et al. (2011), which emphasize that a feature is a system property that is relevant to some stakeholder. None of them is perfectly suited for a strictly formalized feature theory. For this reason, a definition is provided that only meets the essential requirements for system composition, decomposition and feature isolation.

Axiom 1. A feature uniquely represents one or more parts of one or more software artifacts.

Axiom 2. A feature must not contain sub-parts that correspond to another feature of the same SPL.

A *feature* is an abstract entity that uniquely represents one or more parts of one or more software artifacts. Axiom 1 completely decouples a feature from its corresponding parts in software artifacts. This allows to focus reasoning on features only when composing systems of a SPL with features, decomposing systems into features, and isolating features. Different types of software artifacts can have a totally different structure. Therefore, dealing with their syntax and semantic must be done on the software artifact side. This is the reason for adapters as for example described in Martinez et al. (2016). It should not be entangled with reasoning about features. This has important consequences. The possibility that two or more features can have common parts in a software artifact is not an issue that needs to be resolved on the side of software artifacts. Rather is must be appropriately dealt with in the feature theory. The same holds for feature interactions. They are not a phenomenon that is specific to software artifacts. Instead, they must be adequately addressed in the feature theory. Only in this way reasoning about features can be effectively separated from the software artifacts.

A very important restriction we place on a part is, that it must not contain any sub-part which is related to more than one feature of the same SPL. That is, there is a one-to-one correspondence between a feature and its associated parts of software artifacts within a specific SPL.

Axiom 2 enforces and complements Axiom 1. It states that a feature cannot be related to (sub-)parts of software artifacts to which another feature of the same SPL is also related. As an important consequence a feature must not exhibit a finer granularity than necessary. This is similar to the concept of a block presented in Martinez et al. (2015). The context that determines the appropriate level of granularity is the given SPL with its systems. This does not exclude that a feature in one SPL is a product of another SPL which means, that itself is composed from features of the other SPL (this conforms to the definition of feature given in Apel et al. (2008)). However, in the context of the given SPL this does not matter.

Given the one-to-one relationship between features and parts of software artifacts it is possible to generally reason about features on an abstract level, without considering the specific software artifacts. All these details of the relationship between a feature and the parts of software artifacts are shown in Fig. 1. As can be seen, a Feature associates one or more Parts, and each Part is always associated with exactly one Feature. A Part belongs to exactly one Software Artifact, as indicated by the filled diamond. A Software Artifact is composed of at least one Part.

These restrictions were formulated in this way because real-world scenarios for feature location can differ enormously. This is also the reason why we refrain from presenting a real-world scenario in this paper, as this would either entail too much complexity or be too contrived.

Another consequence of Axioms 1 and 2 is that a feature must be principally locatable. Therefore, it is necessary that a situation exists that allows this feature to be effectively isolated. If such a situation cannot be arranged in principle, it is pointless to assume the existence of this feature and trying to locate it.

A central assumption of our feature definition is that a feature has exactly the "right" granularity with respect to a particular SPL and the systems available for it. Cutting this feature into sub features would not give a valid result, since these sub features could not be isolated with the given SPL and the systems available for it. If the feature were of larger granularity, it did include parts that are associated to other features as well. Hence, it would also be impossible to isolate it effectively, because this attempt did include other features as well.

For these reasons it depends on the particular SPL and the systems given for it whether a feature can be isolated at all or not. If the SPL and its available systems do not allow to isolate this feature in principle, it cannot exist because it is impossible to locate it. This is different from the situation that the particular SPL and its given systems principally allow to isolate a feature, but evaluating the isolation gives an empty result. In this case, the feature could principally exist but it does not. In practice, this conclusion is less strong, since the chosen representation of a software artifact or the implementation of the feature location process may be imperfect.

As soon as a feature has been located it does exist. That is, *located* and *exists* are synonyms for the same feature state. If a feature is not located in a situation that principally allows to isolate it, it does also not

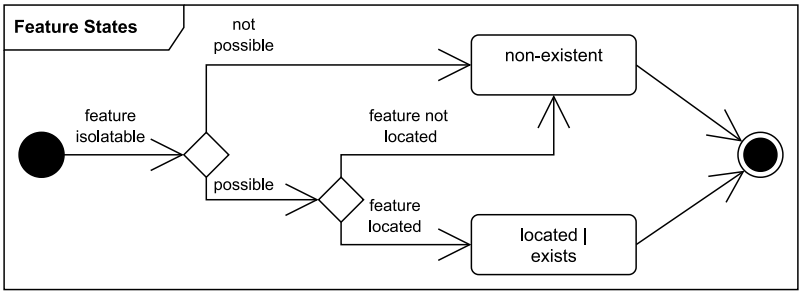


Fig. 2. Possible feature states after isolation.

exist. The state diagram in Fig. 2 summarizes the possible feature states. It should be noted that the diagram describes exactly one situation in which a number of systems of a SPL and a certain set of candidate features are given. In addition, it may be possible to create a different situation with additional systems of the SPL that allow to isolate a feature. However, this is not shown in the diagram.

The existence of a feature and the possibility to isolate it principally depends on a particular SPL and its available systems. As will be demonstrated in Section 3.4.2 the presence or absence of a single system can enable or disable the possibility that a feature can be isolated and thus exists.

2.3. Feature categories

The following feature categories include only features that are associated with one or more parts of one or more software artifacts. All types of features that do not fulfill this requirement are excluded, for example, abstract features. The purpose of the categories is to elevate the observable phenomena on the side of software artifacts to a formal level that abstracts from their concrete representation. They are also necessary for comparison with other approaches of feature location (see Section 4.2). Furthermore, knowledge of these categories and the exact number of features they contain is essential for validation (see Section 3). The corresponding terms for these categories used in the ArgoUML SPL benchmark are or-traces, and-traces, and not-traces (Martinez et al., 2018a).

- Two or more features can have commonality in their corresponding parts of software artifacts (or-traces).
- Two or more features may require the addition of specific parts that take care for the interaction of the features that are present in a system (and-traces).
- If a feature is not included in a system, a kind of stub or substitute feature can be required that has to be included because of the absence of the particular feature (not-traces).

Principally, feature interactions between features and not-features can exist. Because of simplicity they are excluded in the following considerations. The primary goal of this section is to address all the included cases described above in the feature theory, so that they all can be consistently formalized and analyzed on the feature level only without considering concrete software artifacts.

The class diagram in Fig. 3 gives an overview of all feature categories that are considered in this paper and their relationships. These include independent features and explicitly dependent features such as or-features, and-features, not-features, or-not-features, and and-not-features. The diagram shows Feature as a root class whose subclass Inherently Dependent Features requires the presence of at least one Independent Feature that is also a subtype of Feature, for example, a Not-Feature can only exist if there is an Independent Feature as a counterpart. The existence of the feature base class allows all features to be treated uniformly. Two possible feature categories are not shown in Fig. 3: commonalities between a Not-Feature and the other Independent Features and feature interactions between a Not-Feature and the other Independent Features. An independent feature can be used in different relationships, for example, it can be specified that the presence of one Independent Feature requires the presence of another. The relationships requires, excludes, inclusive-or, and exclusive-or are shown because they are widely used in feature diagrams (Thüm et al., 2011; Seidl et al., 2016).

2.3.1. Independent features

The fundamental category of features are independent features. An *independent feature* is a feature which can principally exist independently of any other feature. An independent feature roughly corresponds to a block in Martinez et al. (2015) on the side of software artifacts. Independent features are uniquely named using the notation $f_1, f_2, f_3, \ldots, f_n$ where the last subscript n denotes the last feature. Let the number of independent features of a SPL be denoted as F.

2.3.2. Inherently dependent features

We can also think of features which can only exist in an inherent dependency on other features. Let the number of these *inherently dependent features* of a SPL be denoted as DF. Next, the various categories of inherently dependent features are described in detail.

2.3.2.1. Or-features. An *or-feature* exists only if at least one of the k features it depends on, exists. k can be any number equal or larger than two up to the total number of independent features F. A feature that is present if at least one of the features f_1, f_2, and f_3 is present is denoted by $f_1 \vee f_2 \vee f_3$. Formula (1) shows how to compute the maximum number of or-features O for F independent features. An or-feature corresponds to commonality in parts of software artifacts that belong to two or more features in other approaches. In Martinez et al. (2015), they are called feature intersections on the side of software artifacts. For the theory presented in this paper it is essential to explicitly qualify and treat a feature intersection as a feature in order to make system composition and decomposition work and to allow for precise feature isolation.

$$O = \sum_{k=2}^{F} \frac{F!}{k! \cdot (F-k)!} \tag{1}$$

2.3.2.2. And-features. An *and-feature* exists only if all k of the features it depends on, exist. k can be any number equal or larger than two up to the total number of independent features F. A feature that is present if and only if features f_1, f_2, and f_3 are present, is denoted by $f_1 \wedge f_2 \wedge f_3$. The number of and-features A is calculated with exactly the same formula as the number of or-features, Formula (1). Therefore, it is not reproduced here. An and-feature corresponds to a feature interaction in other work. For the purpose of this paper, it is treated as a feature of its own.

We call the number of independent features involved in an or-feature or an and-feature *interaction order* or even shorter *order*. This term is also used in Apel et al. (2013) for the same purpose but defined as the number of involved features minus 1. In Section 4.1 R will be introduced as a symbol for denoting the interaction order.

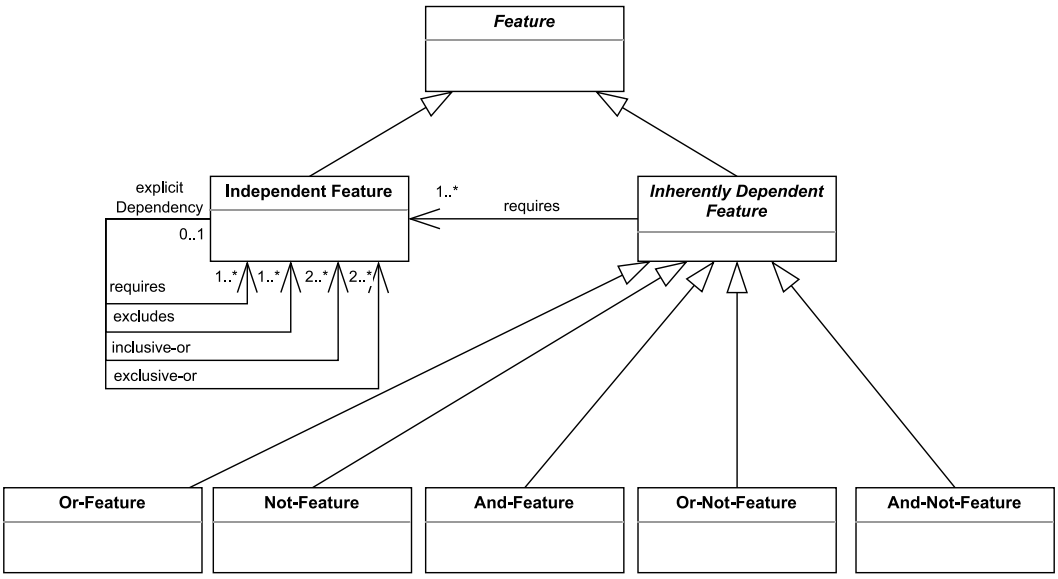


Fig. 3. Feature categories and their relationships.

2.3.2.3. Not-features. A *not-feature* exists only if the feature it does depend on is not an element of a system. The unique name of a not-feature results from placing a ¬ in front of the name of the feature it depends on. So $\neg f_1$ is the not-feature for f_1, which only occurs if f_1 is not present in a system. For example $S_1 = \{f_1\}, S_2 = \{\neg f_1\}$. We could now assume, that a system may or must contain an endless number of not-features. This is not the case, because a not-feature can only exist, if the feature it depends on exists in at least one system of the SPL. That is, $\neg f_1$ requires f_1 to exist. If there is no f_2 there principally cannot be a $\neg f_2$. There may arise the impression that independent features and their not-features are symmetric. But this is not true in every respect because a not-feature always depends on an independent feature but not vice versa. If F is the number of independent features there are also $N = F$ possible not-features. A not-feature corresponds to parts of a software artifact that show up, when a particular feature is not included in a system. It could be discussed, whether it is meaningful to consider this case at all. In classic feature diagrams it could be also modeled as an xor-group with two features of an abstract parent feature (see Fig. 4 (b)). Nevertheless, this case is included because it occurs for example in the ArgoUML SPL benchmark (Martinez et al., 2018a). There, the location of so-called not-traces is required.

2.3.2.4. Or-not-features. An *or-not-feature* exists only if at least one of the k not-features it depends on, exists. k can be any number equal or larger than two up to the total number of corresponding not-features N. An or-not-feature that is present if at least one of the not-features $\neg f_1$, $\neg f_2$, and $\neg f_3$ is present is denoted by $\neg f_1 \vee \neg f_2 \vee \neg f_3$. It must be emphasized that ¬ has a higher priority than ∨ here. The number of or-not-features ON is also calculated according to Formula (1).

2.3.2.5. And-not-features. An *and-not-feature* exists only if all k of the not-features it depends on exist. k can be any number equal or larger than two up to the total number of corresponding not-features N. A feature, which is present if and only if features $\neg f_1$, $\neg f_2$, and $\neg f_3$ are present, is denoted by $\neg f_1 \wedge \neg f_2 \wedge \neg f_3$. The number of and-not-features AN is also calculated according to Formula (1).

2.3.2.6. Excluded and impossible features. As mentioned earlier, commonalities and interactions between not-features and the remaining independent features are in principle possible and may occur but have been excluded for simplicity. Features that represent commonalities between not-features and the remaining independent features

Table 1
Overview of feature category formulas (excluding ONF and ANF).

Feature category	Formula
Independent **F**eature	F
Or-Feature	$O = \sum_{k=2}^{F} \frac{F!}{k!\cdot(F-k)!}$
And-Feature	$A = O$
Not-Feature	$N = F$
Or-**N**ot-Feature	$ON = O$
And-**N**ot-Feature	$AN = O$
Inherently **D**ependent **F**eature	$DF = O + A + N + ON + AN$
Total number of Features	$T = F + DF = 2 \cdot F + 4 \cdot \sum_{k=2}^{F} \frac{F!}{k!\cdot(F-k)!}$

are called ONF which is a contraction of ON (or-not-features) and F (independent features). Their number is calculated with Formula (2).

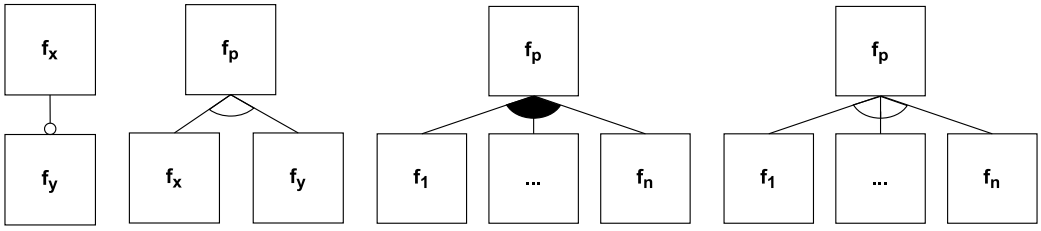
$$ONF = \sum_{k=1}^{F} \frac{F!}{k! \cdot (F-k)!} - 1 \tag{2}$$

Features that represent interaction between not-features and the remaining independent features are called ANF which is a contraction of AN (and-not-features) and F (independent features). Their number is equal to ONF. Actually, ONF and ANF should be considered in the calculation of the inherently dependent features DF and thus also in the total number of features T. However, since they are not considered in the programs and only marginally in this text, they are not listed in Table 1.

Or- and and-features can only exist if the corresponding independent features are present. There is no dependency among or- or and-features. Not-features can only exist, if there are independent features. Or-not- or and-not-features can only exist if there are not-features. Thus, there is an indirect dependency of or-not- or and-not-features on independent features. There is no dependency among or-not- or and-not-features.

Furthermore, each inherently dependent feature must be in principle a feature that can be isolated. None of the inherently dependent features increases the number of (possible) systems of a SPL. This number only depends on the number of independent features F. Table 1 summarizes the feature categories including their calculation.

The decision to exclude ONF- and ANF-features is intended to limit a further increase in the number of feature categories to be considered. In the same way, or-not- and and-not-features or even all



(a) Requires relationship

(b) Excludes relationship

(c) Inclusive-or group

(d) Exclusive-or group

Fig. 4. Restrictions among independent features.

feature categories involving not-features could be excluded. For this reason, Axiom 3 reflects an arbitrary, but conscious decision that can be revised in the future.

Axiom 3. The above system for categorizing features is complete, that is, there are no additional categories besides independent features, or-features, and-features, not-features, or-not-features, and and-not-features.

2.3.3. Explicitly dependent features

Having F totally independent features there are 2^F possible different systems. Now, we can introduce restrictions among independent features turning some of them into *explicitly dependent features*. These relationships are summarized in Fig. 4 and include requires, excludes, inclusive-or group, and exclusive-or group. It uses the classic notation of feature diagrams to illustrate a few specific situations that may occur in practice. The theory presented in this paper does not use feature diagrams for illustration.

2.3.3.1. Requires relationship. A first restriction is that feature f_y *requires* feature f_x. This can be expressed in various ways. One possibility is to use propositional logic, that is $f_y \rightarrow f_x$ (Thüm et al., 2011). In a classic feature diagram, there are two ways to depict that. First, one can draw an edge between a parent feature f_x and its child feature f_y as shown in Fig. 4(a). Another option is to use a textual notation to record this dependency.

2.3.3.2. Excludes relationship. A second restriction is that feature f_x excludes feature f_y (and vice versa). In propositional logic this can be described as $(f_x \vee f_y) \wedge \neg(f_x \wedge f_y)$. In a classic feature diagram are several ways to depict that. If f_x and f_y are child features of the same parent feature f_p, they can be connected by an empty arc, which usually denotes a so-called exclusive-or feature group as shown in Fig. 4(b). Another option is to use a textual notation to record this dependency.

2.3.3.3. Inclusive-or group. There is an inclusive-or relationship between a parent feature f_p and a set of child features $f_1 \ldots f_n$ if one or more child features can be selected when the parent feature is part of the product. In propositional logic this can be described as $f_1 \vee \ldots \vee f_n \Leftrightarrow f_p$. A filled arc is used to denote the inclusive-or feature group as shown in Fig. 4(c).

2.3.3.4. Exclusive-or group. There is an exclusive-or relationship between a parent feature f_p and a set of child features $f_1 \ldots f_n$ if only one child feature can be selected when the parent feature is part of the product. In propositional logic this can be described as $(f_1 \vee \ldots \vee f_n \Leftrightarrow f_p) \wedge \bigwedge_{i<j} \neg(f_i \wedge f_j)$. An empty arc is used to denote the exclusive-or feature group as shown in Fig. 4(d).

It has to be pointed out that requires or excludes relationships, that is, explicitly dependent features, can only exist between independent features. They must not introduce contradictions. A consequence of having explicitly dependent features is that an explicitly dependent feature cannot be isolated in each situation that principally allows its isolation. But nonetheless, it must be possible to arrange a special situation that allows to isolate and thus to locate it.

The existence of an explicitly dependent feature can be only inferred for a SPL and its available systems. Only if we have firm knowledge that the number of available systems is exhaustive, it can be concluded that a feature is explicitly dependent.

The existence of explicitly dependent features reduces the number of possible different systems in a SPL. Because explicit dependencies can be introduced arbitrarily among features, it is not possible to give a general formula for calculating the number of possible systems in a SPL with explicitly dependent features.

2.4. System composition

In a given system, a feature is either present or not. Therefore, we have decided to use sets to abstractly represent systems with features. Moreover, set theory provides all the necessary operations to reason about features and systems, that is, to compose and decompose systems and to isolate and subsequently to locate features. For example, Ziadi et al. (2012), Acher et al. (2013) and Apel et al. (2013) also use sets for the same or similar purposes.

From the perspective of features a *system* (synonyms are *product* or *variant*) can be uniquely described by enumerating its features (Thüm et al., 2011; Ziadi et al., 2012), that is $\{f_1, f_2\}$, $\{f_1, f_3\}$, $\{f_1, f_2, f_3\}$ and so on. These system descriptions use mathematical sets. A feature corresponds to an element of such a set. Each unique system of a SPL is referred to by S followed by a numerical subscript. Thus, all systems are named S_1, S_2, S_3, …, S_n. Here, the last subscript n denotes the number of systems. The number of features and the number of systems are different and should not be confused. If two sets describing systems are equal, it is the same system. A set united with itself, results in the same set. Furthermore, a set intersected with itself, results in the same set. All independent features can be combined in all possible ways. That is, if there are F independent features, the number of different systems can be calculated with Formula (3). All possible different systems can be considered as SPL. Let the number of possible systems of a SPL be denoted by S.

$$S = 2^F \tag{3}$$

If explicitly dependent features exist in a SPL, the number of possible systems must be smaller than S. Then, the actual number depends on the explicitly dependent features and can be calculated precisely only if all explicit dependencies are known. For example, as pointed out by Acher et al. (2013), the number of configurations is exponential to the number of features, which here corresponds to Formula (3). Translating feature models into logical representations allows enumerating configurations and counting configurations. If one considers explicitly

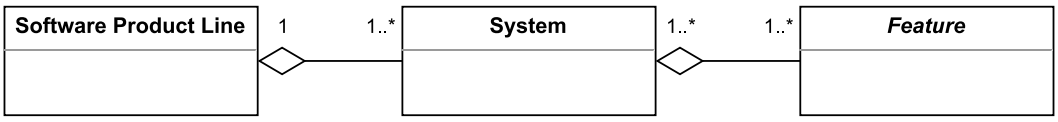


Fig. 5. Relations between SPL, system, and feature.

Table 2
Overview of the various models describing a SPL and their composition by feature categories.

Model	Constituting feature categories
$M1$	F
$M2$	$F+O$
$M3$	$F+A$
$M4$	$F+O+A$
$M5$	$F+N$
$M6$	$F+O+N$
$M7$	$F+A+N$
$M8$	$F+O+A+N$
$M9$	$F+N+ON$
$M10$	$F+N+AN$
$M11$	$F+O+N+ON$
$M12$	$F+A+N+ON$
$M13$	$F+O+A+N+ON$
$M14$	$F+O+N+AN$
$M15$	$F+A+N+AN$
$M16$	$F+O+A+N+AN$
$M17$	$F+O+N+ON+AN$
$M18$	$F+A+N+ON+AN$
$M19$	$F+O+A+N+ON+AN=F+DF=T$

dependent features, this must be done similarly in our approach. In contrast, the existence of dependent features cannot change the number of possible configurations, as will be explained in the next section.

The class diagram in Fig. 5 gives an overview of the relations between SPL, system, and feature. As can be seen, a SPL aggregates at least one System, and a System aggregates at least one Feature.

2.5. Models M1–M19

Depending on F, feature isolation can become computationally intensive – and feature location even more because of large software artifacts with a complex structure. Therefore, it is necessary to provide guidelines which features of which category should be located with a higher priority. For this purpose, a series of so-called models is proposed.

A model captures either knowledge or assumptions about the feature categories that make up a SPL. Each model is a semantically valid combination of feature categories. The first group of models ($M1$–$M4$) does not contain any feature category related to not-features, while the second group of models ($M5$–$M19$) does. Furthermore, the models are structured in terms of the number of feature categories involved, taking into account the dependencies between the different feature categories.

Table 2 shows how the various models describing a SPL can be composed by feature categories. The numbering of the models is somewhat arbitrary. It must be pointed out, that this table only includes models with semantically valid combinations of feature categories, for example, $F+N$. Models with combinations that are per se invalid, are not included, for example, $F+ON$. If it is known that not-features do not exist, but it is assumed that or- or and-features can exist, the systems of a SPL can be composed according to one of the models $M1$, $M2$, $M3$, or $M4$. Therefore, only attempts to locate the appropriate features are required. Isolating and then locating or-features of the lowest order is the most plausible step. If an or-feature of lower order cannot be located it is pointless trying to locate any higher order or-features in which the corresponding features are involved. For example, if $f_1 \vee f_2$ cannot be located, it is pointless trying to locate $f_1 \vee f_2 \vee f_3$. Next, and-features can be isolated and located. Unfortunately, higher order and-features exist completely independent of respective lower-order and-features. For example, if neither $f_1 \wedge f_2$, $f_1 \wedge f_3$, nor $f_2 \wedge f_3$ can be located, it is still possible that $f_1 \wedge f_2 \wedge f_3$ can be successfully located. The findings from Apel et al. (2013) suggest that the number of feature interactions diminishes with respect to their order. Therefore, it may be advisable trying to locate only lower order and-features, and continue the location of higher order and-features only, if there is a strong guess that they are relevant and therefore could be located.

All programs used for the theoretical validation (see Section 3) take each of these models into account. Their execution provides exactly the same results for each model. Only a few selected models are explained in more detail in this paper.

Feature location is an iterative process with two activities. First, it is checked whether a certain feature can be isolated at all, assuming F independent features and considering the available systems of a SPL. The attempt to isolate a feature does not include the processing of the available software artifacts for the purpose of feature location. If the situation does not allow to isolate the specific feature, the process terminates. If the feature can be isolated in principle, the available artifacts must be analyzed to locate the feature, which is the second activity. If the result is empty and errors in the processing of the software artifacts can be ruled out or have a low probability, the specific feature does not exist because it cannot be located and the process terminates. If the result is not empty, the feature could be successfully located (again, the caveat of imperfect processing applies) and the process also terminates. As has already been done implicitly before, the term *isolation* is used in the following when the focus is on the first activity, while the term *location* refers to location but also includes isolation as a necessary preceding step. For example, if no not-feature can be located, trying to locate or-not- or and-not-features is pointless.

Not-features are redundant because they can be replaced by explicitly dependent features. That is, features f_1 and $\neg f_1$ can be replaced by f_1 and f_2 plus the excludes relationship $(f_1 \vee f_2) \wedge \neg(f_1 \wedge f_2)$. The addition of the excludes relationship is necessary to limit the number of possible systems to 2, which equals the number of systems resulting from f_1 and $\neg f_1$. Therefore, Formula (3) changes to $S = 2^{2 \cdot F}$ when not-features are completely replaced by explicitly dependent features. In addition, F exclude relationships must be managed separately to keep the number of possible systems the same as for Formula (3) without the previous change. The decision to allow not-features or not does not change the total number of features that can be isolated in principle (provided ANF and ONF are added to T in this case). The advantage of using not-features is that the number of possible systems is only the square root of the case without them. Thus, there is a trade-off between the presence or absence of not-related features that is difficult to decide in principle. Since the ArgoUML SPL benchmark explicitly requires locating not-traces (Martinez et al., 2018a), not-features are also considered here.

Tables 3 and 4 show a complete example for a SPL with three independent features f_1, f_2, f_3 with respect to $M19$. Table 3 lists all categories, the features they can (in principle) contain, and the number of features per category. Table 4 lists the definitions of all systems by specifying their (possible) features and their number. The number of different systems in a SPL according to Formula (3) is $S = 2^3 = 8$.

Table 3
All feature categories for SPL with three independent features.

Feature category	Features	Count
F	f_1, f_2, f_3	3
O	$f_1 \vee f_2, f_1 \vee f_3, f_2 \vee f_3, f_1 \vee f_2 \vee f_3$	4
A	$f_1 \wedge f_2, f_1 \wedge f_3, f_2 \wedge f_3, f_1 \wedge f_2 \wedge f_3$	4
N	$\neg f_1, \neg f_2, \neg f_3$	3
ON	$\neg f_1 \vee \neg f_2, \neg f_1 \vee \neg f_3, \neg f_2 \vee \neg f_3, \neg f_1 \vee \neg f_2 \vee \neg f_3$	4
AN	$\neg f_1 \wedge \neg f_2, \neg f_1 \wedge \neg f_3, \neg f_2 \wedge \neg f_3, \neg f_1 \wedge \neg f_2 \wedge \neg f_3$	4
DF	$O + A + N + ON + AN$	19
T	$F + DF$	22

Table 4
All eight systems of a SPL with three independent features according to model $M19$.

System	Features	Count
S_1	$\neg f_1, \neg f_2, \neg f_3, \neg f_1 \vee \neg f_2, \neg f_1 \vee \neg f_3, \neg f_2 \vee \neg f_3, \neg f_1 \vee \neg f_2 \vee \neg f_3, \neg f_1 \wedge \neg f_2, \neg f_1 \wedge \neg f_3, \neg f_2 \wedge \neg f_3, \neg f_1 \wedge \neg f_2 \wedge \neg f_3$	11
S_2	$f_1, \neg f_2, \neg f_3, \neg f_2 \vee \neg f_3, \neg f_2 \wedge \neg f_3$	5
S_3	$f_2, \neg f_1, \neg f_3, \neg f_1 \vee \neg f_3, \neg f_1 \wedge \neg f_3$	5
S_4	$f_3, \neg f_1, \neg f_2, \neg f_1 \vee \neg f_2, \neg f_1 \wedge \neg f_2$	5
S_5	$f_1, f_2, f_1 \vee f_2, f_1 \wedge f_2, \neg f_3$	5
S_6	$f_1, f_3, f_1 \vee f_3, f_1 \wedge f_3, \neg f_2$	5
S_7	$f_2, f_3, f_2 \vee f_3, f_2 \wedge f_3, \neg f_1$	5
S_8	$f_1, f_2, f_3, f_1 \vee f_2, f_1 \vee f_3, f_2 \vee f_3, f_1 \vee f_2 \vee f_3, f_1 \wedge f_2, f_1 \wedge f_3, f_2 \wedge f_3, f_1 \wedge f_2 \wedge f_3$	11

2.6. Feature isolation

Feature isolation is the necessary prerequisite for feature location. It attempts to isolate a feature for a particular SPL with a given number of independent features and its available systems. The result of feature isolation is a set expression that tells us, if a specific feature can be effectively isolated so that it can be located. If this is the case, the resulting set expression has to be evaluated with the adequately represented software artifacts of the available systems. The representation of a software artifact depends on its structure and usually differs for each software artifact type. For example, an adequate representation of source code is an abstract syntax tree for a programming language. Adequate representations of software artifacts are not in the focus of this paper. In Müller and Eisenecker (2019) a Java source code parser is used to obtain a representation of Java source code programs as a graph resembling an abstract syntax tree that is stored in a graph database.

If we want to locate a feature f_h, we must isolate it. This is done by intersecting all (existing/available) system descriptions which contain the desired feature f_h. From the result we subtract the union of all (existing/available) system descriptions which do not contain the desired feature f_h. A system description is a set that contains the names of all features that are present in the particular system. These operations are summarized in Formula (4) which has been presented in Müller and Eisenecker (2019). A related procedure using elementary model construction primitives is presented as Algorithm 1 in Ziadi et al. (2012). Please note, that Formula (4) applies to any feature of any category.

$$\{f_h\} = \bigcap S_{f_h \in S} \setminus \bigcup S_{f_h \ni S} \tag{4}$$

If the resulting set is empty, f_h does not exist. If the resulting set has exactly one element this is f_h, that is, f_h was effectively isolated and can thus be located. If the resulting set has more than one element, among them f_h, then f_h could not be isolated. There are two reasons for this: (a) f_h cannot be isolated in principle, because it does not meet Axiom 2, (b) the especially arranged situation does not allow to effectively isolate f_h, that is, there are systems missing either in the left or the right part of set subtraction. If (b) is not a valid option, (a) is the only valid option. This statement is theoretically validated in Section 3.4.3.

Next, we derive a formula that allows to compute how many different instances of this formula exist for a given number of different systems. If there are S systems the left operand may be the empty set, while the right operand comprises S sets. This can be permuted until the left operand comprises S sets, and the right operand is the empty set. This can be calculated using the binomial coefficient $\binom{S}{k} = \sum_{k=0}^{S} \frac{S!}{k!\cdot(S-k)!}$. S means here the number of available systems, and k means the sample size. Actually, this formula simplifies to Formula (5).

$$D = 2^S \tag{5}$$

A SPL with $F = 3$ independent features results in $S = 2^3 = 8$ different systems. Applying Formula (5) results in $D = 2^8 = 256$ different instances of the feature isolation expression in Formula (4). This number also includes nonsensical expressions that contain f_h in the intersection part and in the union part.

3. Validation

The validation of the approach is performed in three different ways. First, each feature is isolated using the system descriptions generated for a SPL according to Formula (4). Second, all expressions with all system descriptions that may isolate features are generated and evaluated. Third, only valid expressions with all system descriptions isolating features are calculated and evaluated. All three approaches produce consistent and correct results. As another aspect of validity the robustness of the approach with respect to feature isolation is tested in several ways. First, a feature is introduced that is not one of the features of a SPL. Second, some system descriptions are eliminated. Third, explicitly dependent features are introduced. Finally, features that are present in every system are introduced. Overall, the results show that the approach is very robust to such perturbations.

3.1. Simulating feature isolation

Based on the definition of a feature in Section 2.2, a simulation of the feature isolation process will be conducted. Its purpose is to validate the soundness of Formula (4) with respect to feature isolation. It does not involve feature location in software artifacts.

To achieve this, first all possible features for a given number of independent features F and a selected model M need to be generated, that is, their names are generated according to the schemata presented in Section 2.3. As a model, $M1$ is the most trivial case because only independent features are considered for system composition and decomposition. Second, all system descriptions (see Section 2.6 for a definition) are generated. This step corresponds to system composition by feature combination. In the third and last step feature isolation is performed. For each feature Formula (4) is built and evaluated. A more detailed workflow for these steps is given at the end of Section 3.1.

Given the way the system descriptions are generated it is obvious that an instantiation of Formula (4) for each feature must exist and give a valid result, because this system was also used to generate the system description. The instantiation of Formula (4) for a feature shows how this feature can be isolated. If for whatever reason, evaluating Formula (4) would result in the empty set, the feature was non-existent.

Based on Formula (4) for isolating a feature it is possible to check whether if the feature can be successfully isolated at all given the available systems, which is discussed in Section 3.4.2. Only by analyzing the feature side – without taking the software artifacts into account – it can be decided whether it is possible at all to isolate the feature. If this is possible, the process of feature location is started. Now the appropriate representations of the software artifacts have to be processed according to the feature-specific instantiation of Formula (4). If the result is empty, the feature is not located and therefore it is non-existent. If there is a result, this is the part or this are the parts of the software artifacts that correspond to the specific feature. In addition, errors in the representation of the software artifacts or the processing of the instantiation of Formula (4) can also lead to an erroneous results.

M19	selected model			
8	T	actual total number of features		
2	F	number of independent features		
6	DF	actual total number of inherently dependent features		
1	O	actual number of or-features		
1	A	actual number of and-features		
2	N	actual number of not-features		
1	ON	actual number of or-not-features		
1	AN	actual number of and-not-features		
4	S	number of systems of SPL		
16	D	number of all set differences of SPL systems		
S1	!f1	!f1 * !f2	!f1 + !f2	!f2
S2	!f1 + !f2	!f2	f1	f1 + f2
S3	!f1	!f1 + !f2	f1 + f2	f2
S4	f1	f1 * f2	f1 + f2	f2
!f1	(S1 & S3) \ (S2 | S4)			
!f1 * !f2	(S1) \ (S2 | S3 | S4)			
!f1 + !f2	(S1 & S2 & S3) \ (S4)			
!f2	(S1 & S2) \ (S3 | S4)			
f1	(S2 & S4) \ (S1 | S3)			
f1 * f2	(S4) \ (S1 | S2 | S3)			
f1 + f2	(S2 & S3 & S4) \ (S1)			
f2	(S3 & S4) \ (S1 | S2)			

Fig. 6. Feature isolation result for two independent features and model $M19$.

Table 5
Mapping between terms in paper and in program.

Term	Paper	Program
Feature	f_1	$f1$
Or-Feature	$f_2 \vee f_3$	$f2 + f3$
And-Feature	$f_2 \wedge f_3$	$f2 * f3$
Not-Feature	$\neg f_4$	$!f4$
System	S_1	$S1$
Collection of systems	(S_4)	$(S4)$
Intersection of systems	$S_1 \cap S_3$	$S1$ & $S3$
Union of systems	$S_2 \cup S_4$	$S2$ | $S4$
Set difference	\	\

The corresponding program, *feature_isolation_demo.cpp*, is included in the GitHub project and fully documented there.[4] It accepts the number of independent features and the chosen model as input and outputs values for the formulas given in Table 1 and Formulas (3) and (5), a list of all systems with names and their defining features, and a list of all features and the set differences that isolate them. Fig. 6 shows the output for $F = 2$ and $M19$.

The reason for selecting $M19$ is, that we want to see system descriptions that include all possible features that can occur which is the most complex case. The motivation for this section is to validate feature isolation by simulating it. Therefore, we do not start with the most simple model, as one would usually do when locating features in practice.

As explained before, the upper part of Fig. 6 shows the various values describing the corresponding SPL, the middle part contains the list with system names plus the features defining each system, and the lower part shows each feature plus the set differences of systems that do isolate it. Since the output is created using ASCII characters only, Table 5 shows how the concepts described before are mapped to the program output.

Before proceeding, a brief outline of the idealized workflow of the program is given.

1. All values, that describe the SPL, are calculated.
2. All features that are available with respect to the selected model are generated.

E1	!f1 * !f2	(S1) \ (S2 | S3 | S4)
E3	!f2	(S1 & S2) \ (S3 | S4)
E5	!f1	(S1 & S3) \ (S2 | S4)
E7	!f1 + !f2	(S1 & S2 & S3) \ (S4)
E8	f1 * f2	(S4) \ (S1 | S2 | S3)
E10	f1	(S2 & S4) \ (S1 | S3)
E12	f2	(S3 & S4) \ (S1 | S2)
E14	f1 + f2	(S2 & S3 & S4) \ (S1)

Fig. 7. Feature differences result for two independent features and model $M19$.

3. All systems that are available for the SPL are generated in terms of the features they contain.
4. For each feature, all systems that do contain it are collected; let this collection be named I, short for Intersection.
5. For each feature, all systems that do not contain it are collected; let this collection be named U, short for Union.
6. Afterwards, each system of I is transformed to a set with the features that define this systems. The resulting feature sets are intersected; let the resulting set be called I'.
7. Additionally, each system of U is transformed into a set with the features that define this system. The resulting feature sets are united; let the resulting set be called U'.
8. Eventually, the set difference of I' and U' is calculated. The resulting set contains the isolated feature as its only element.

This is already sufficient to effectively isolate features. Nevertheless, we will go one step further.

3.2. Generating all possible set differences

The program *feature_differences_demo.cpp*, which is also part of the GitHub project, generates all possible differences of system sets, regardless of whether a system contains a particular feature or not.[4] If the corresponding set difference expression is non-empty, it is memorized. The program calculating all possible set differences does this in a systematic way. Conceptually, it uses a bit string with as many elements as the number of systems of a given SPL. Initially, only the least significant bit of the bit string is set. In the following iteration this bit string is incremented by 1, that is, the natural number that it represents is incremented by 1. This is repeated until the largest number that can be represented with this bit string minus 1 is reached. This procedure generates all possible bit patterns with two exceptions, namely the bit pattern with no bit set, and the bit pattern with all bits set. Such a bit string is now transformed into a difference of systems, that possibly isolates a feature. Whenever a bit is set, the corresponding system is included in the collection of systems that will be intersected later, namely I. Each bit, that is not set, is included in the collections of systems that will be united afterwards, namely U. This is the reason, why the bit strings with no bit set and with all bits set are excluded, because the resulting differences must be the empty set in these two cases. Nevertheless, in Section 3.5 a valid use case for all systems being included in I will be presented. Now, all systems of I and U are transformed into sets I' and U' that contain the corresponding features. Finally, the set difference of I' and U' is calculated. If it is non-empty it is stored along with the bit string that represents the given set difference of sets of systems.

The output of this program is almost the same as that of the first program. The only difference is that in the list of set differences a first column is added that contains the ID of the difference that represents the specific set difference. This ID begins with the letter E, short for Expression, to calculate the set difference, followed by the decimal value that represents the bit string of the set difference.

Fig. 7 shows the output for two independent features and model $M19$. Since the general information and the system descriptions are identical to that of Fig. 6, only the lower part is reproduced.

S1			
S2	f1		
S3	f2		
S4	f1	f2	
S5	f3		
S6	f1	f3	
S7	f2	f3	
S8	f1	f2	f3
E170	f1	(S2 & S4 & S6 & S8) \ (S1 | S3 | S5 | S7)	
E204	f2	(S3 & S4 & S7 & S8) \ (S1 | S2 | S5 | S6)	
E240	f3	(S5 & S6 & S7 & S8) \ (S1 | S2 | S3 | S4)	

Fig. 8. Feature differences result for three independent features and model $M1$.

Table 6
Independent features: Expression IDs, isolated features, and the corresponding bit string.

Expression ID	Feature name	Bit string
$E170$	f_1	10101010
$E204$	f_2	11001100
$E240$	f_3	11110000

Now, we will take a closer look at these difference expressions. To do this, the second program is executed for three independent features and model $M1$. The middle and the lower part of the corresponding output is shown in Fig. 8.

Interestingly, eight systems are reported for the SPL, despite $S1$ does not contain any feature. Since the model does not include not-features, we can assume that $S1$ is a truly empty system. A system that does not contain any feature is in contradiction to Fig. 5, because there a system is connected with at least one feature. It can therefore be assumed that this is only a hypothetical system which cannot actually exist. Nevertheless, there are three valid expressions, each isolating one of the available features.

Next, each difference is represented as a bit string. Since there are eight systems, the bit string contains 8 Bit. Table 6 shows the expression IDs plus the isolated features and the corresponding bit string as a pattern with 1 representing a set bit, and 0 representing a bit that is not set. A bit that is set represents a system that is included in I, which is the left operand of the set difference expression, and a bit that is not set represents a system that is included in U, which is the right operand of the set difference expression.

The resulting bit strings exhibit a systematic pattern. Since a feature is present or not, exactly half of the systems must contain it, and the other half not. Additionally, all combinations of independent features being present, must exist. Therefore, the top most halves of systems are each partitioned into two more halves, one that contains another feature and one that does not. This is repeated, until each half contains only one feature, as it is the case for feature f_1.

3.2.1. Isolating independent features

The corresponding values of the expression IDs are computed by interpreting the bit pattern as binary natural numbers and transforming them into decimal numbers by applying Horner's scheme.

This suggests, that it is possible to systematically compute each expression that can isolate 1 of n independent features. Indeed, there are various ways to achieve this. Two possibilities for calculating the difference expression to isolate an independent feature will be explained below and sample implementations will be given using pseudo code.

The first possibility assumes that each system is represented as an unsigned integer value. In this paper, systems are numbered from 1 to S. But the lower bound of unsigned integers is 0. For this reason 1 must be subtracted from the variable that represents a system. Next, this value is combined by bitwise *and* with a bit mask that represents the feature of interest. This bit mask results from shifting a set least significant bit (LSB) to the left by the value of the feature of interest minus 1. Running the loop variable `feature` from 0 to F - 1 is essential, because a set LSB already represents a set feature. Listing 1 shows a pseudo code fragment that implements the afore-mentioned algorithm.

```
print "Feature Name";
printTab;
printLine "Bit String";
for (unsigned_int feature = 0; feature < F; ++feature)
begin
  print "f";
  print feature + 1; printTab;
  for (unsigned_int system = S; system > 0;
       system = system - 1)
  begin
    if ((system - 1) bitAnd (1 leftShift feature) <> 0)
    then print 1
    else print 0;
  end
  printLine;
end
```

Listing 1: First variant for calculating difference expressions

Overall, this program performs $F \cdot S$ calculations. This number cannot be reduced since all systems must be considered for each feature. It is limited by the number of bits of the largest integer type on the given platform. In principle, calculating the difference expressions required to isolate six independent features, the largest integer type must have 64 bit. For many platforms, this corresponds to the number of bits of the largest unsigned integer type.

Next, a program with a higher limit is presented. Besides `feature` and `system` three more variables are needed. Initially, the bool variable `bit` is initialized to true. The integer variable `stride` is initialized to 1. It is the number of steps that have to be passed before flipping the content of `bit`. When `bit` is sent to output, its values are automatically converted to 0 for false and 1 for true. 1 is the first value that is output for the first system, because it does include the first feature. Initially, `stride` is 1. Therefore, `bit` is flipped immediately. For each next feature the value of `stride` is doubled. Thus, for the second feature `stride` is 2, for the third feature it is 4, and so on. Inside the first for loop the integer variable `counter` is defined and initialized to 0. For each iteration of the loop over the number of systems `counter` is incremented by 1. In the inner loop it is checked if `counter` equals `stride`. If so, the content of `bit` is flipped and `counter` is reset to 0. The pseudo code in Listing 2 implements this algorithm.

```
print "Feature Name";
printTab;
printLine "Bit String";
boolean bit = true;
unsigned_int stride = 1;
for (unsigned_int feature = 0; feature < F;
     feature = feature + 1)
begin
  print "f";
  print (feature + 1);
  printTab;
  unsigned_int counter = 0;
  for (unsigned_int system = 0; system < S;
       system = system + 1)
  begin
    print bit;
    counter = counter + 1;
    if (counter == stride)
    begin
      bit = not bit;
      counter = 0;
    end
  end
  printLine;
  stride = stride * 2;
end
```

Listing 2: Second variant for calculating difference expressions

Table 7
Not-features: Expression IDs, isolated features, and the corresponding bit string.

Expression ID	Feature name	Bit string
$E15$	$\neg f_3$	00001111
$E51$	$\neg f_2$	00110011
$E85$	$\neg f_1$	01010101

Table 8
Or-features: Expression IDs, isolated features, set differences, and bit strings.

Expr. ID	Feature name	Set difference	Bit string
$E170$	$f1$	$(S2\ \&\ S4\ \&\ S6\ \&\ S8) \setminus (S1 \mid S3 \mid S5 \mid S7)$	10101010
$E204$	$f2$	$(S3\ \&\ S4\ \&\ S7\ \&\ S8) \setminus (S1 \mid S2 \mid S5 \mid S6)$	11001100
$E238$	$f1+f2$	$(S2\ \&\ S3\ \&\ S4\ \&\ S6\ \&\ S7\ \&\ S8) \setminus (S1 \mid S5)$	11101110
$E240$	$f3$	$(S5\ \&\ S6\ \&\ S7\ \&\ S8) \setminus (S1 \mid S2 \mid S3 \mid S4)$	11110000
$E250$	$f1+f3$	$(S2\ \&\ S4\ \&\ S5\ \&\ S6\ \&\ S7\ \&\ S8) \setminus (S1 \mid S3)$	11111010
$E252$	$f2+f3$	$(S3\ \&\ S4\ \&\ S5\ \&\ S6\ \&\ S7\ \&\ S8) \setminus (S1 \mid S2)$	11111100
$E254$	$f1+f2+f3$	$(S2\ \&\ S3\ \&\ S4\ \&\ S5\ \&\ S6\ \&\ S7\ \&\ S8) \setminus (S1)$	11111110

This approach is also limited by the type of the largest unsigned int which must allow to support the number of possible systems as computed according to Formula (3). On many platforms, the size of the largest **unsigned int** is 8 byte. In this case, the maximum number of features is $F = 64$. These limits can be an argument for including not-related-features when possible. When mapping not-features to inherently dependent features (plus exclude relationships), the limits are reduced to $F = 3$ (algorithm in Listing 1) and $F = 32$ (algorithm in Listing 2), respectively.

Using one of these algorithms, each of the F expressions for isolating independent features can be calculated in a time proportional to S. It is no longer necessary to calculate all possible differences, which is also nonsensical, because this includes many invalid difference expressions that contain a specific system S in the I and in the U part. Moreover, calculating all possible 2^S set differences instead of only F causes a long runtime for even moderate numbers of independent features such as $F = 5$.

The expressions for isolating features of other categories can also be calculated systematically, as shown below.

3.2.2. Isolating not-features

The most simple procedure is the one for isolating not-features. For this category, the negated bit strings for isolating independent features give the desired results. Table 7 shows this for three independent features and model $M5$, that includes independent features and not-features only.

Obviously, the bit pattern for $\neg f_i$ results from the bitwise negation of the bit pattern for f_i.

3.2.3. Isolating or-features

To explain the algorithm for calculating the set differences of or-features, three independent features and model $M2$ are used. This model comprises independent features and or-features. Table 8 shows the resulting set differences including difference expression IDs, feature names, and bit strings. Looking at the set differences in this table gives no direct clue how to calculate the set differences for or-features. This changes when looking at the corresponding bit strings. Again, a set bit in the bit string indicates a system that is included into the left operand I of the difference expression, and an unset bit indicates a system that is included in the right operand U of the difference expression.

Now, the scheme for calculating the bit strings for or-features can be easily recognized. To obtain the bit string that represents a specific or-feature we combine the bit strings of all related independent features by bitwise *or*, as Table 9 shows.

By applying Horner's scheme to the resulting bit strings the decimal values of the expression IDs can be calculated, which is not shown here.

Table 9
Scheme for calculating the bit strings for or-features.

	$f_1 \vee f_2$	$f_1 \vee f_3$	$f_2 \vee f_3$	$f_1 \vee f_2 \vee f_3$
f_1	10101010	10101010		10101010
f_2	11001100		11001100	11001100
f_3		11110000	11110000	11110000
	11101110	11111010	11111100	11111110

Table 10
And-features: Expression IDs, isolated features, set differences, and bit strings.

Expr. ID	Feature name	Set difference	Bit string
$E128$	$f1*f2*f3$	$(S8) \setminus (S1 \mid S2 \mid S3 \mid S4 \mid S5 \mid S6 \mid S7)$	10000000
$E136$	$f1*f2$	$(S4\ \&\ S8) \setminus (S1 \mid S2 \mid S3 \mid S5 \mid S6 \mid S7)$	10001000
$E160$	$f1*f3$	$(S6\ \&\ S8) \setminus (S1 \mid S2 \mid S3 \mid S4 \mid S5 \mid S7)$	10100000
$E170$	$f1$	$(S2\ \&\ S4\ \&\ S6\ \&\ S8) \setminus (S1 \mid S3 \mid S5 \mid S7)$	10101010
$E192$	$f2*f3$	$(S7\ \&\ S8) \setminus (S1 \mid S2 \mid S3 \mid S4 \mid S5 \mid S6)$	11000000
$E204$	$f2$	$(S3\ \&\ S4\ \&\ S7\ \&\ S8) \setminus (S1 \mid S2 \mid S5 \mid S6)$	11001100
$E240$	$f3$	$(S5\ \&\ S6\ \&\ S7\ \&\ S8) \setminus (S1 \mid S2 \mid S3 \mid S4)$	11110000

Table 11
Scheme for calculating the bit strings for and-features.

	$f_1 \wedge f_2$	$f_1 \wedge f_3$	$f_2 \wedge f_3$	$f_1 \wedge f_2 \wedge f_3$
f_1	10101010	10101010		10101010
f_2	11001100		11001100	11001100
f_3		11110000	11110000	11110000
	10001000	10100000	11000000	10000000

Table 12
Or-not-features: Expression IDs, isolated features, set differences, and bit strings.

Expr. ID	Feature name	Set difference	Bit string
$E15$	$!f3$	$(S1\ \&\ S2\ \&\ S3\ \&\ S4) \setminus (S5 \mid S6 \mid S7 \mid S8)$	00001111
$E51$	$!f2$	$(S1\ \&\ S2\ \&\ S5\ \&\ S6) \setminus (S3 \mid S4 \mid S7 \mid S8)$	00110011
$E63$	$!f2+!f3$	$(S1\ \&\ S2\ \&\ S3\ \&\ S4\ \&\ S5\ \&\ S6) \setminus (S7 \mid S8)$	00111111
$E85$	$!f1$	$(S1\ \&\ S3\ \&\ S5\ \&\ S7) \setminus (S2 \mid S4 \mid S6 \mid S8)$	01010101
$E95$	$!f1+!f3$	$(S1\ \&\ S2\ \&\ S3\ \&\ S4\ \&\ S5\ \&\ S7) \setminus (S6 \mid S8)$	01011111
$E119$	$!f1+!f2$	$(S1\ \&\ S2\ \&\ S3\ \&\ S5\ \&\ S6\ \&\ S7) \setminus (S4 \mid S8)$	01110111
$E127$	$!f1+!f2+!f3$	$(S1\ \&\ S2\ \&\ S3\ \&\ S4\ \&\ S5\ \&\ S6\ \&\ S7) \setminus (S8)$	01111111

3.2.4. Isolating and-features

To describe the algorithm for calculating the set differences of and-features we proceed similarly. First, this is done for three independent features and model $M3$. This model comprises independent features and and-features. Table 10 shows the resulting set differences including difference expression IDs, features, and bit strings.

Now, the algorithm for calculating the bit strings for and-features becomes obvious. To obtain the bit string that represents a specific and-feature the bit strings of all related independent features must be combined by bitwise *and*, as Table 11 shows.

Again, Horner's scheme for converting the resulting bit strings into the decimal values of the expression IDs is omitted.

3.2.5. Isolating or-not-features

The set difference for isolating or-not-features is similar to that of the corresponding calculation for or-features. The difference is, that the bit strings of the corresponding not-features have to be combined by bitwise *or*. Table 12 shows the set differences and bit strings for not- and or-not-features for three independent features and model $M9$. This model contains independent features, not-features, and or-not-features. Table 13 shows the corresponding calculation scheme.

3.2.6. Isolating and-not-features

The set difference for isolating and-not-features is calculated by combining the bit strings of the corresponding not-features by bitwise *and*. Table 14 shows the set differences and bit strings for not- and and-not-features for three independent features and model $M10$. This model contains independent features, not-features, and and-not-features. Table 15 shows the corresponding calculation scheme.

Table 13
Scheme for calculating the bit strings for or-not-features.

	$\neg f_1 \vee \neg f_2$	$\neg f_1 \vee \neg f_3$	$\neg f_2 \vee \neg f_3$	$\neg f_1 \vee \neg f_2 \vee \neg f_3$
$\neg f_1$	01 010 101	01 010 101		01010101
$\neg f_2$	00 110 011		00 110 011	00110011
$\neg f_3$		00 001 111	00 001 111	00001111
	01 110 111	01 011 111	00 111 111	01111111

Table 14
And-not-features: Expression IDs, isolated features, set differences, and bit strings.

Expr. ID	Feature name	Set difference	Bit string
$E1$	$!f1 * !f2 * !f3$	$(S1) \setminus (S2 \mid S3 \mid S4 \mid S5 \mid S6 \mid S7 \mid S8)$	00000001
$E3$	$!f2 * !f3$	$(S1 \,\&\, S2) \setminus (S3 \mid S4 \mid S5 \mid S6 \mid S7 \mid S8)$	00000011
$E5$	$!f1 * !f3$	$(S1 \,\&\, S3) \setminus (S2 \mid S4 \mid S5 \mid S6 \mid S7 \mid S8)$	00000101
$E15$	$!f3$	$(S1 \,\&\, S2 \,\&\, S3 \,\&\, S4) \setminus (S5 \mid S6 \mid S7 \mid S8)$	00001111
$E17$	$!f1 * !f2$	$(S1 \,\&\, S5) \setminus (S2 \mid S3 \mid S4 \mid S6 \mid S7 \mid S8)$	00010001
$E51$	$!f2$	$(S1 \,\&\, S2 \,\&\, S5 \,\&\, S6) \setminus (S3 \mid S4 \mid S7 \mid S8)$	00110011
$E85$	$!f1$	$(S1 \,\&\, S3 \,\&\, S5 \,\&\, S7) \setminus (S2 \mid S4 \mid S6 \mid S8)$	00001111

Table 15
Scheme for calculating the bit strings for and-not-features.

	$\neg f_1 \vee \neg f_2$	$\neg f_1 \vee \neg f_3$	$\neg f_2 \vee \neg f_3$	$\neg f_1 \vee \neg f_2 \vee \neg f_3$
$\neg f_1$	01 010 101	01 010 101		01010101
$\neg f_2$	00 110 011		00 110 011	00110011
$\neg f_3$		00 001 111	00 001 111	00001111
	00 010 001	00 000 101	00 000 011	00000001

3.2.7. Isolating features of ONF and ANF categories

In Section 2.3.2.6, we mention feature categories that involve independent and not-related features, namely ONF and ANF. To calculate the bit strings representing the set differences to isolate them, the bit-strings for isolating the involved independent features and not-features must be combined with bitwise *or* for ONF, and with bitwise *and* for ANF.

3.3. Calculating only valid set differences

Based on the insights before, a third program (*feature_calculation_demo.cpp*) is implemented that takes exactly the same input as the two preceding programs and produces exactly the same output as the second program. But it does this by calculating only valid set differences.

3.4. Testing the robustness of the approach

Next, we will investigate how resistant the algorithm described in Formula (4) behaves against disturbances including spurious features and missing systems. Finally, we will show how it handles explicitly dependent features.

3.4.1. Spurious features

One possible disturbance is the introduction of a spurious feature. Its name is chosen after the *spurious interrupt* for which no source can be found. This spurious feature, it will be denoted with f_x, may appear more or less systematically in any system of a SPL. In the following we will examine the effect of various occurrences of f_x in a SPL with three independent features and model $M1$. The results are already shown above in Fig. 8. The expression IDs that isolate f_1, f_2 and f_3 are $E170$, $E204$ and $E240$. They have been shown several times.

First, we assume that f_x appears exclusively in S_1. Next, we show $E170$ and its substitution by sets containing f_x that define the corresponding systems. Afterwards, the resulting set expression is evaluated.

$$
\begin{aligned}
E170 =& (S_2 \cap S_4 \cap S_6 \cap S_8) \setminus (S_1 \cup S_3 \cup S_5 \cup S_7) \\
\rightarrow& (\{f_1\} \cap \{f_1, f_2\} \cap \{f_1, f_3\} \cap \{f_1, f_2, f_3\}) \\
& \setminus (\{f_x\} \cup \{f_2\} \cup \{f_3\} \cup \{f_2, f_3\}) \\
\rightarrow& \{f_1\} \setminus \{f_x, f_2, f_3\} \\
\rightarrow& \{f_1\}
\end{aligned}
$$

Evaluating the left operand of the set difference, that is I, already gives the final result which is $\{f_1\}$. The spurious feature f_x is included in the right operand, that is U, only, but has no effect on the final result.

In $E204$ and $E240$ system S_1 also appears as part of the right operand of the set difference, that is U. Hence, the presence of f_x cannot have any effect onto the result. Since S_1 appears only in U, let us now introduce f_x into S_8, which is a system that appears only in the left operand of the set difference, that is in I. In I, each system is replaced by the features that it can contain, and afterwards these feature sets are intersected. Since f_x appears only in S_8, it cannot be an element of the resulting intersection. Therefore, f_x cannot have any influence on the result of evaluating each set difference for independent features.

Even if f_x were introduced in three of the four systems that are part of I, f_x would not change the result of evaluating I because of set intersection.

Finally, f_x is introduced in S_2, S_4, S_6, and S_8. Now, the presence of f_x must have an influence when evaluating $E170$. Next, the same steps that have been applied before, are repeated again.

$$
\begin{aligned}
E170 =& (S_2 \cap S_4 \cap S_6 \cap S_8) \setminus (S_1 \cup S_3 \cup S_5 \cup S_7) \\
\rightarrow& (\{f_1, f_x\} \cap \{f_1, f_2, f_x\} \cap \{f_1, f_3, f_x\} \cap \{f_1, f_2, f_3, f_x\}) \\
& \setminus (\{\} \cup \{f_2\} \cup \{f_3\} \\
& \cup \{f_2, f_3\}) \\
\rightarrow& \{f_1, f_x\} \setminus \{f_2, f_3\} \\
\rightarrow& \{f_1, f_x\}
\end{aligned}
$$

Evaluating I gives a result set that contains f_1 and f_x. But, what does this mean? In this case, f_1 and f_x cannot be distinguished any longer, that is, both features collapse. With respect to the feature definition introduced in this paper, it is not possible to effectively isolate f_1 or f_x. Therefore, both features are in fact the same feature that can be denoted by f_1.

In expressions $E204$ and $E240$, f_x would even not appear in the results of evaluating I, since it is not a feature in all systems that are used for I.

The last possibility is to introduce f_x in each of the eight systems S_1 to S_8. In this case, f_x would always be an element of the set that results from evaluating each left operand, that is I, of the set difference. But it would also be an element of the set that results from evaluating each right operand, that is U, of the set difference. But then, it must be eliminated by evaluating each $I \setminus U$. Therefore, even if f_x were present in each system of the SPL it could not have any influence on isolating one of the independent features.

Until now, various possibilities for introducing a spurious feature f_x into a SPL containing independent features only have been considered. Does it make a difference if f_x were introduced in systems of a SPL that can contain any kind of dependent features? The answer is no. If the systems of a SPL can contain any kind of features, this does not change the number of systems. Only the number of set differences increases by exactly the number of dependent features that can be extracted in principle. In general, there are again four possible cases.

1. If f_x is only part of all systems that appear on the left side of a set difference, that is I, f_x cannot be distinguished from the feature to be isolated.
2. If f_x is only part of any number of systems but not all systems that appear in I, f_x becomes eliminated when evaluating I.
3. If f_x is only part of any number of systems that appear on the right side of a set difference, that is U, it cannot have any influence because it can only be subtracted when evaluating the corresponding set difference.

4. If f_x is a feature of all systems, it would appear in the set resulting from the evaluation of I as well as in the result set of evaluating U, and thus be removed when evaluating the set difference.

3.4.2. Missing systems

All differences of feature sets of systems presented above always include all systems of the SPL. In the following, the consequences will be analyzed if one or more systems of a SPL are not available. First, three independent features for model $M1$ will be considered, see Fig. 8 for details.

If only S_1 is not available, this has no effect at all, because S_1 is empty and appears only as part of U. If S_1 and S_2 are not available, only $E170$ can be affected, but not $E204$ and $E240$, because here S_1 and S_2 occur only in U. This is different for $E170$, because here, S_2 is part of I. Therefore, $E170$ is stepwise evaluated, but without S_1 and S_2 being present.

$$\begin{aligned}E170 =&(S_4 \cap S_6 \cap S_8) \setminus (S_3 \cup S_5 \cup S_7)\\ &\rightarrow(\{f_1,f_2\} \cap \{f_1,f_3\} \cap \{f_1,f_2,f_3\}) \setminus (\{f_2\} \cup \{f_3\} \cup \{f_2,f_3\})\\ &\rightarrow\{f_1\} \setminus \{f_2,f_3\}\\ &\rightarrow\{f_1\}\end{aligned}$$

As we can see, the absence of S_1 and S_2 has no effect on the successful isolation of f_1. As soon as I is evaluated, the desired result is available.

Now, the situation without S_1, S_2, and S_3 is considered. This does not affect the isolation of f_1, because S_1 and S_3 appear only as parts of U, while S_2 not being available as part of I does not prevent the successful isolation of f_1. That is different for $E204$, because here, S_3 is part of I. Therefore, the consequences for evaluating $E204$ will be examined, again by stepwise evaluation.

$$\begin{aligned}E204 =&(S_4 \cap S_7 \cap S_8) \setminus (S_5 \cup S_6)\\ &\rightarrow(\{f_1,f_2\} \cap \{f_2,f_3\} \cap \{f_1,f_2,f_3\}) \setminus (\{f_3\} \cup \{f_1,f_3\})\\ &\rightarrow\{f_2\} \setminus \{f_1,f_3\}\\ &\rightarrow\{f_2\}\end{aligned}$$

Actually, systems S_1, S_2, and S_3 not being available does not prevent the successful isolation of f_2. Again, the desired result is available directly after the evaluation of I. $E240$ requires no detailed analysis, because here, the absent systems would only occur in U.

Next, we assume that systems S_1, S_2, S_3, and S_4 are not available. $E240$ is still not affected, because the absent systems would only be part of U. But this is different for $E170$ and $E204$. In each of these expressions, there are two missing systems in I. First, $E170$ will be examined by stepwise evaluation.

$$\begin{aligned}E170 =&(S_6 \cap S_8) \setminus (S_5 \cup S_7)\\ &\rightarrow(\{f_1,f_3\} \cap \{f_1,f_2,f_3\}) \setminus (\{f_3\} \cup \{f_2,f_3\})\\ &\rightarrow\{f_1,f_3\} \setminus \{f_2,f_3\}\\ &\rightarrow\{f_1\}\end{aligned}$$

This is the first time that U becomes relevant. The correct result is not given before evaluating the set difference. Second, $E204$ will be evaluated stepwise.

$$\begin{aligned}E204 =&(S_7 \cap S_8) \setminus (S_5 \cup S_6)\\ &\rightarrow(\{f_2,f_3\} \cap \{f_1,f_2,f_3\}) \setminus (\{f_3\} \cup \{f_1,f_3\})\\ &\rightarrow\{f_2,f_3\} \setminus \{f_1,f_3\}\\ &\rightarrow\{f_2\}\end{aligned}$$

Again, evaluating the set difference gives the correct result. Next, systems S_1 through S_5 will be removed. Now, I of all set expressions changes. First, $E170$ is evaluated stepwise.

$$\begin{aligned}E170 =&(S_6 \cap S_8) \setminus (S_7)\\ &\rightarrow(\{f_1,f_3\} \cap \{f_1,f_2,f_3\}) \setminus (\{f_2,f_3\})\\ &\rightarrow\{f_1,f_3\} \setminus \{f_2,f_3\}\\ &\rightarrow\{f_1\}\end{aligned}$$

Interestingly, there is no impact on obtaining the correct result, which is available right after the evaluation of the set difference. Now, the same is done for $E204$.

$$\begin{aligned}E204 =&(S_7 \cap S_8) \setminus (S_6)\\ &\rightarrow(\{f_2,f_3\} \cap \{f_1,f_2,f_3\}) \setminus (\{f_1,f_3\})\\ &\rightarrow\{f_2,f_3\} \setminus \{f_1,f_3\}\\ &\rightarrow\{f_2\}\end{aligned}$$

Again, the correct result is available after evaluating the set difference. Eventually, $E240$ is stepwise evaluated.

$$\begin{aligned}E240 =&(S_6 \cap S_7 \cap S_8) \setminus ()\\ &\rightarrow(\{f_1,f_3\} \cap \{f_2,f_3\} \cap \{f_1,f_2,f_3\}) \setminus \{\}\\ &\rightarrow\{f_3\} \setminus \{\}\\ &\rightarrow\{f_3\}\end{aligned}$$

After evaluating I, the correct result is given. Now, S_1 through S_6 will be removed. Following, each difference expression is evaluated stepwise.

$$\begin{aligned}E170 =&(S_8) \setminus (S_7)\\ &\rightarrow(\{f_1,f_2,f_3\}) \setminus (\{f_2,f_3\})\\ &\rightarrow\{f_1\}\end{aligned}$$

Again, the correct result is available after evaluating the set difference. Next, $E204$ for isolating f_2 is evaluated.

$$\begin{aligned}E204 =&(S_7 \cap S_8) \setminus ()\\ &\rightarrow(\{f_2,f_3\} \cap \{f_1,f_2,f_3\}) \setminus \{\}\\ &\rightarrow\{f_2,f_3\} \setminus \{\}\\ &\rightarrow\{f_2,f_3\}\end{aligned}$$

In this case, it happens the first time, that f_2 cannot be successfully isolated. Finally, $E240$ for isolating f_3 is evaluated.

$$\begin{aligned}E240 =&(S_7 \cap S_8) \setminus ()\\ &\rightarrow(\{f_2,f_3\} \cap \{f_1,f_2,f_3\}) \setminus \{\}\\ &\rightarrow\{f_2,f_3\} \setminus \{\}\\ &\rightarrow\{f_2,f_3\}\end{aligned}$$

Here, it happens again that a feature, in this case f_3, cannot be successfully isolated. Furthermore, we could analyze other variations of missing systems, and we could investigate the consequences of absent systems when trying to isolate dependent features. But this will not be done here. This topic will be resumed in Section 4.1. Overall, it is remarkable, that the algorithm for isolating features is relatively robust with respect to missing systems.

3.4.3. Explicitly dependent features

Explicitly dependent features, such as $f_2 \rightarrow f_1$ or $(f_1 \vee f_2) \wedge \neg(f_1 \wedge f_2)$, reduce the number of available systems, as pointed out in Section 2.3.3.

First, the case of f_2 requiring f_1 is considered for three independent features and model $M1$. In the following, all systems are deleted that are invalid with respect to the aforementioned constraint.

$$\begin{aligned}S_1 =&\{\}\\ S_2 =&\{f_1\}\\ S_4 =&\{f_1,f_2\}\\ S_5 =&\{f_3\}\\ S_6 =&\{f_1,f_3\}\end{aligned}$$

$S_8 = \{f_1, f_2, f_3\}$

This reduces the number of available systems to six. Next, each difference expression is evaluated stepwise for examining the consequences of removing systems S_3 and S_7. It should be emphasized, that this is a variation of missing systems. First, $E170$ is evaluated stepwise.

$$\begin{aligned} E170 &= (S_2 \cap S_4 \cap S_6 \cap S_8) \setminus (S_1 \cup S_5) \\ &\rightarrow (\{f_1\} \cap \{f_1, f_2\} \cap \{f_1, f_3\} \cap \{f_1, f_2, f_3\}) \setminus (\{\} \cup \{f_3\}) \\ &\rightarrow \{f_1\} \setminus \{f_2, f_3\} \\ &\rightarrow \{f_1\} \end{aligned}$$

After evaluating I, f_1 has been successfully isolated. Now, we proceed with $E204$.

$$\begin{aligned} E204 &= (S_4 \cap S_8) \setminus (S_1 \cup S_2 \cup S_5 \cup S_6) \\ &\rightarrow (\{f_1, f_2\} \cap \{f_1, f_2, f_3\}) \setminus (\{\} \cup \{f_1\} \cup \{f_3\} \cup \{f_1, f_3\}) \\ &\rightarrow \{f_1, f_2\} \setminus \{f_1, f_3\} \\ &\rightarrow \{f_2\} \end{aligned}$$

After evaluating the set difference, f_2 has been successfully isolated. Now, the stepwise evaluation of $E240$ follows.

$$\begin{aligned} E240 &= (S_5 \cap S_6 \cap S_8) \setminus (S_1 \cup S_2 \cup S_4) \\ &\rightarrow (\{f_3\} \cap \{f_1, f_3\} \cap \{f_1, f_2, f_3\}) \setminus (\{\} \cup \{f_1\} \cup \{f_1, f_2\}) \\ &\rightarrow \{f_3\} \setminus \{f_1, f_2\} \\ &\rightarrow \{f_3\} \end{aligned}$$

After evaluating I, f_3 has been successfully isolated. Despite the fact, that each independent feature can be successfully isolated, we cannot unambiguously state that f_2 explicitly depends on f_1 in this case. Only if we knew for sure, that the given number of systems is exhaustive, this would be a valid conclusion.

Second, the case of $(f_1 \vee f_2) \wedge \neg(f_1 \wedge f_2)$ is considered. What follows next is the list of systems that are available given this restriction.

$$\begin{aligned} S_1 &= \{\} \\ S_2 &= \{f_1\} \\ S_3 &= \{f_2\} \\ S_5 &= \{f_3\} \\ S_6 &= \{f_1, f_3\} \\ S_7 &= \{f_2, f_3\} \end{aligned}$$

Now, systems S_4 and S_8 are no longer present. Next, each difference expression is evaluated stepwise.

$$\begin{aligned} E170 &= (S_2 \cap S_6) \setminus (S_1 \cup S_3 \cup S_5 \cup S_7) \\ &\rightarrow (\{f_1\} \cap \{f_1, f_3\}) \setminus (\{\} \cup \{f_2\} \cup \{f_3\} \cup \{f_2, f_3\}) \\ &\rightarrow \{f_1\} \setminus \{f_2, f_3\} \\ &\rightarrow \{f_1\} \end{aligned}$$

Evaluating I successfully isolates f_1.

$$\begin{aligned} E204 &= (S_3 \cap S_7) \setminus (S_1 \cup S_2 \cup S_5 \cup S_6) \\ &\rightarrow (\{f_2\} \cap \{f_2, f_3\}) \setminus (\{\} \cup \{f_1\} \cup \{f_3\} \cup \{f_1, f_3\}) \\ &\rightarrow \{f_2\} \setminus \{f_1, f_3\} \\ &\rightarrow \{f_2\} \end{aligned}$$

Evaluating I results in f_2.

$$\begin{aligned} E240 &= (S_5 \cap S_6 \cap S_7) \setminus (S_1 \cup S_2 \cup S_3) \\ &\rightarrow (\{f_3\} \cap \{f_1, f_3\} \cap \{f_2, f_3\}) \setminus (\{\} \cup \{f_1\} \cup \{f_2\}) \\ &\rightarrow \{f_3\} \setminus \{f_1, f_2\} \\ &\rightarrow \{f_3\} \end{aligned}$$

Table 16
Systems with corresponding features of a SPL with five independent features and model $M1$.

System	Features
S_1	f_4, f_5
S_2	f_1, f_4, f_5
S_3	f_2, f_4, f_5
S_4	f_1, f_2, f_4, f_5
S_5	f_3, f_4, f_5
S_6	f_1, f_3, f_4, f_5
S_7	f_2, f_3, f_4, f_5
S_8	f_1, f_2, f_3, f_4, f_5

Evaluating I gives f_3. Again, all independent features can be isolated successfully. But as in the preceding case it cannot be validly concluded, that f_1 and f_2 are mutually exclusive features. Only the firm knowledge, that the SPL does not comprise more than the given systems could support this conclusion.

The cases of the inclusive-or group (Fig. 4(c)) and the exclusive-or group (Fig. 4(d)) are not examined in detail here.

3.5. Core

Next, we consider a SPL with five independent features, two of them being present in all systems, namely f_4 and f_5. They can be assumed to depend on the presence of none or any number of all other features. Features such as f_4 and f_5 are sometimes referred to as SPL core (Martinez et al., 2018a). Table 16 lists all eight systems of the SPL for model $M1$. A closer look at this table reveals that S_1 contains f_4 and f_5. The intersection I of all systems also gives a result set, that contains f_4 and f_5. In this case, no system is left for being united, that is U is empty. Evaluating $I \setminus U$ also gives the result set with f_4 and f_5 as elements. The algorithm outlined in Section 2.6 (Formula (4)) can be applied to effectively isolate features f_1, f_2, and f_3, as it was demonstrated in Section 3.4.1 where f_x was introduced as spurious feature into each system of the SPL.

Furthermore, it would be a valid transformation to subtract S_1 from each system, before resulting in $S'_1 = \{\}, S'_2 = \{f_1\}$, and so on. Applying Formula (4) to the resulting S' systems would also give valid results. This step would be an appropriate optimization before performing feature location with large amounts of data.

As outlined above f_4 and f_5 cannot be isolated because they are contained in each system of the SPL. Therefore, f_4 and f_5 collapse to a single feature, that can be isolated by intersecting all available systems of the SPL. To infer that f_4 and f_5 form the SPL core is only valid if we know for sure that there exist no other systems beside S_1 to S_8. One approach to nevertheless isolate f_4 and f_5 is outlined in Section 4.2 below.

If the systems of the SPL could also contain inherently dependent features this would not influence the isolation of any feature that inherently depends on f_1, f_2 or f_3. As pointed out in Section 4.2 inherently dependent features cannot be isolated from features that are always present in a SPL. With the exception of any category related to not-features they collapse with the features being always present.

4. Future perspectives

Obviously, the expressions that isolate features can be evaluated in different ways. The possible advantages of optimizing feature isolation by specializing the evaluation in terms of the available system descriptions or the order of the isolated features should be explored. Feature location is a particular challenge when only one system of a SPL (usually with all features) is available. Existing work shows promising results, but also points to some problems. It should be investigated whether the presented approach can solve these open issues.

4.1. Further potential of set differences

The preceding Sections 3.4.2 and 3.4.3 emphasize the need for a further analysis of the set difference expressions for isolating features. First, it can be observed, that for independent features and not-features each part of a difference, I and U, involves 2^{n-1} systems, that is, exactly half of the systems is to be intersected, and half of the systems is to be united. In Section 3.4.2, it was shown that in some cases not all of the systems of a SPL have to be available for successfully isolating an independent feature or a not-feature. In several cases evaluating only the left part I of the set difference already gives the desired result. If the number of available systems decreases further, evaluating the U part of the set difference results in the features that have to be subtracted from the left part to successfully isolate the feature.

This is different for or-features and and-features. Since all or-features appear in a system, that are related to a specific independent feature as soon as this independent feature is included in the system, the number of systems to be intersected increases with the interaction order of the specific dependent or-feature. The number of systems to be united decreases correspondingly. This is also true for or-not-features.

The situation is inversely symmetric for and-features. An and-feature will only show up, if all related independent features are present in a system. Therefore, the number of systems included in I decreases with an increasing interaction order of and-features, while the number of systems included in U decreases correspondingly. This is also true for and-not-features.

Precisely, be R (short for Rank) the order of the interaction ($R \leq F$) between features. As pointed out in Section 2.3.2.2, we define R as the number of features that are involved in an interaction, while Apel et al. (2013) define R as the number of involved features minus 1. For R interacting or-features the number of systems including these features is calculated as $S-2^{(F-R)}$. Correspondingly, the number of systems that do not contain R interacting or-features is $2^{(F-R)}$. For and-features, the number of systems containing R interacting and-features equals $2^{(F-R)}$. Correspondingly the number of systems that do not contain them is $S-2^{(F-R)}$. The numbers of interacting or-not- or and-not-features are calculated accordingly.

We assume that there is a systematic relation between available systems and features that can be successfully isolated. Knowing this relation can be exploited in two different ways. First, it can be precisely determined if a specific feature can be isolated given a lower number of available systems of a SPL at all. Second, it can be used for optimizing the evaluation of set difference expressions within various respects. For example, the evaluation can be stopped as soon as the interesting feature is isolated. Possibly the order of evaluating the sub-expressions could be optimized as well. All this is not covered in this paper and will be subject of future research.

4.2. Analyzing a single system

The approach for feature location described in this paper crucially depends on the one-to-one correspondence between a feature and its associated parts of software artifacts (Axiom 1), refer to Fig. 1 for details. Now, let us assume, that there is only one system of a SPL that includes all independent features and possibly dependent features as well. In this case it is not possible to perform an analysis based on the proposed approach. But this is not a deficit of this approach, this is simply a consequence of the fact, that there is only one system available. If it were possible to create additional artifacts for this single system that are related to the features to be located in the necessary way, the described approach can be applied to these artifacts.

Exactly this is the focus of Michelon et al. (2021). In this approach multiple artificial system variants from the execution of a single system are created. In combination with static code analysis they are reused for feature location afterwards. The authors report *"From the results, we can see that our technique for locating features might not be effective in large systems as in small ones, mainly when most of the lines of source code belong to common implementations, i.e. the feature base. In our study, this leads to many false negatives in ArgoUML"*. These lines of source correspond to or-features in the theory presented in this paper. For a single system containing all (independent) features it is impossible to locate and-features, not-features, or-not-features, or and-not-features, because locating these features requires systems that do not contain all independent features. Combining the approach of Michelon et al. (2021) and the theory presented in this paper could allow for a more precise feature location having only one system plus execution traces available and give a sound understanding of what exactly has been located.

Table 17
Limitations.

F	T	S	T · S
1	2	2	4
2	8	4	32
3	22	8	176
4	52	16	832
5	114	32	3 648
6	240	64	15 360
7	494	128	63 232
8	1004	256	257 027
9	2026	512	1 037 312
10	4072	1024	4 169 728

5. Limitations

Table 17 shows the total number of features T for model $M19$ and the number of systems for F independent features ranging from 1 to 10 plus the product of T and S. T for $M19$ is calculated according to the last formula listed in Table 1.

According to Formula (3), S grows exponentially depending on F. Obviously, T grows even faster (but still exponentially in terms of algorithmic complexity). Computing and outputting the difference expressions that isolate each feature that can principally exist according to $M19$ creates high memory demands for increasing values of F. Storing only the bit strings representing the difference expressions for $F = 10$ in $M19$ would require 4.169.728 byte, if each bit were output as ASCII-character '0' or '1' (not including line feeds). One could argue that this is a waste of memory, because the data could be saved in binary format, thus requiring 509 kibibyte only. But this also does not scale to considerably larger values of F. The theory presented in this paper itself is not a scalable algorithm to compute all possible features for any F for $M19$. Instead it serves as a basis for developing algorithms for purposefully locating features.

All information that is available for a given SPL and systems available for it must be taken into account when locating features. A few examples will be outlined below.

1. Assumed a SPL has F independent features and n ($0 < n < F$) of them are surely known to be present in each system of the SPL. According to Section 3.4.3 the available systems can be transformed such that only the features that actually vary are analyzed. This is done by computing the SPL core first (intersection of all available systems) and subtract it from each system that is available. The actual feature location analysis takes place afterwards.
2. If it is known for sure that a SPL does not contain not-features, no features that depend on not-features need to be isolated.
3. If a location analysis results in n not-features, only for the n not-features or-not-features or and-not-features can be isolated in principle.
4. When locating or-features, this should be done incrementally starting with the lowest order. If $f_1 \vee f_2$ cannot be observed, it can be validly concluded that any higher order dependent features involving f_1 and f_2 do not exist.

5. For a given SPL, available systems and firm knowledge about its features it can be analyzed whether a partial evaluation of a set difference for isolating a specific feature is sufficient. In Section 3.4.2 it is shown that partial evaluations can give the desired result. Moreover, before conducting the feature location with the actual software artifacts, it can be precisely determined which analyses can give results, in which order with respect to efficiency features should be located, and which feature location attempts must fail because of missing systems.

Hence, just calculating everything that can be calculated in principle is not a well-thought option.

Everything related to feature isolation is on a formal level. When it comes to feature location, further constraints have to be considered. First, adequate representations of software artifacts must be available. Second, these artifacts must be stored in a way that allows to efficiently process them further. Third, the processing of the actual data must be carefully planned and executed afterwards. This can be done in practice, as already demonstrated in Müller and Eisenecker (2019).

The fourth program (*fl.cpp*) only requires memory for the difference expressions that are specifically calculated. This reduces the required memory at the expense of increased execution time, since difference expressions can be computed repeatedly. Nevertheless, this program is effectively reduced to compute the difference expressions for F features (always assuming model $M19$) and outputting them to specific files. After starting the program, it asks for a value for F. Upon its completion, it creates six files containing difference expressions to isolate the features of each category from Table 1. This program also contains a member function (`bool difference_expression_generator::operator()(maxnat_t f, maxnat_t s) const`) that allows to calculate the bit value for feature `f` and system `s`, with the exemplar of the `difference_expression_generator` initialized to F. This program can serve as a starting point for further analysis.

6. Related work

First, related work on feature modeling is reported. This includes references to feature definitions including the properties of features, feature categories, the relationship of features to software artifacts, feature interactions, and feature models as a means for software composition and reasoning about SPLs. Many of the referenced approaches are more or less dependent on concrete software artifacts. Next, work about feature location is addressed. A variety of different techniques is used to identify and locate features. A number of approaches also use sets and set operations (usually on representations of specific software artifacts) for the purpose of feature location.

6.1. Feature modeling

The variability among products of a SPL can be modeled using feature models – which is focused in this paper – orthogonal variability models or decision models (Raatikainen et al., 2019).

Kang et al. (1990) initially introduced feature models in the context of Feature-Oriented Domain Analysis (FODA). Feature models are trees or directed acyclic graphs comprising common and variable features of the SPL (Benavides et al., 2010). Features that can appear independently of other features in systems of a SPL are referred to as independent features in this work. These basic feature models include relationships such as *mandatory*, *optional*, *alternative*, and *or* as well as cross-tree relationships such as *requires* and *excludes*. In this paper, we call these relationships between independent features *explicitly dependent features*.

Several extensions of the original FODA notation have been proposed (Czarnecki et al., 2004; Schobbens et al., 2006) including *feature cardinalities* (Czarnecki et al., 2002), *groups and group cardinalities* (Czarnecki, 1999; Riebisch et al., 2002; Czarnecki et al., 2005), *attributes* (Czarnecki et al., 2002), *relationships* (Kang et al., 1998; Griss et al., 1998; Lee et al., 2002; van Gurp et al., 2001), *feature categories and annotations* (Griss et al., 1998; Czarnecki and Eisenecker, 2000; Czarnecki, 1999), *aspectual features* (Zaatar and Hamza, 2011), and *abstract features* (Thüm et al., 2011).

The theory presented in this paper focuses on the formal aspects of features with respect to their number and the given systems for a SPL without considering the implementation of software artifacts and their structure. There are no abstract features, and each feature has a cardinality of 0 or 1. There are two fundamental categories of features, namely independent features and dependent features. Propositional formulas can be used to describe constraints between independent features that turn some of them into explicitly dependent features. A dependent feature can only exist if the independent features it depends on exist. This way, an aspectual feature in classic feature models can be adequately replaced with an independent feature plus corresponding and-features as dependent features. Feature interactions that are reflected in the representation of software artifacts can be replaced by and- or or-features.

Apel et al. (2008) introduce an algebra for features and feature composition, where a feature consists of one or more source code artifacts, that is, features can be structured. This is different from our approach that strictly distinguishes between a feature and its associated part(s) of software artifacts (see Fig. 1). This provides the necessary formal basis for effective reasoning about features and the systems of a SPL without the need to immediately consider the representation of a feature in software artifacts and their structure.

Acher et al. (2013) present the domain-specific language FAMILIAR providing support for separating concerns, reasoning, and scripting facilities for the large scale management of feature models. In addition, FAMILIAR provides operators for creating, counting, and checking the validity of configurations based on constraints between features formulated in propositional logic. A configuration is the counterpart of a system description in our approach.

6.2. Feature location

To foster research in the field of feature location for SPLs, there are publicly available benchmarks. Apart from the ArgoUML SPL benchmark (Martinez et al., 2018a), there are the Linux-Kernel-based benchmark (Xing et al., 2013) and the Eclipse Feature Location Benchmark (EFLBench) (Martinez et al., 2018b).

There are many feature location approaches in the context of SPLs using textual, static, dynamic, or combined strategies (Cruz et al., 2019). Rubin and Chechik (2013) describe popular static and dynamic feature location techniques and discuss their suitability for SPLs. Assunção and Vergilio (2014) and Assunção et al. (2017) provide mapping studies on feature location for SPL migration and on reengineering legacy applications into SPLs. Cruz et al. (2019) conduct a literature review on feature location in SPLs and compare three feature location techniques using the ArgoUML SPL.

Ziadi et al. (2012) use sets and set operations to locate features, but they focus on UML diagrams reverse engineered from source code. Nie and Zhang (2012) present an approach based on Latent Dirichlet Allocation (LDA) using textual and structural information. Rubin and Chechik (2012) compute diff sets by comparing the source code of product variants with different features to each other and propose the two heuristics filtering and score modification to improve feature location. Xue et al. (2012) exploit commonalities and differences of product variants by software differencing and Formal Concept Analysis (FCA) techniques to improve feature location with Latent Semantic Indexing (LSI). Al-Msie'deen et al. (2013b) introduce an approach also based on FCA. Al-Msie'deen et al. (2013a) extend it with lexical and structural similarity measures computed by LSI. Linsbauer et al. (2014) combine the approaches Configuration-Aware Program Analysis (CAPA) and Extraction and Composition for Clone-and-Own (ECCO) (Linsbauer

et al., 2013; Fischer et al., 2014) for feature location. They analyze the source code of product variants and create conditional System Dependence Graphs (SDGs). Then they apply an algorithm to map features to source code where feature traces are identified based on the differences between product variants. Other than the theory presented here, Fischer et al. (2014) directly analyze concrete software artifacts. In this paper, a base module refers to artifacts that implement a feature without feature interaction, which roughly corresponds to an independent feature in our approach, and a derivative module refers to artifacts that implement a feature interaction, which roughly corresponds to a dependent feature in our approach. ECCO also uses set operations, for example for detecting superfluous code. Michelon et al. (2019) use this approach to contribute to the ArgoUML SPL Benchmark. Kästner et al. (2014) present a semi-automatic approach for variability mining to support feature location. Martinez et al. (2015) introduce Bottom-Up Technologies for Reuse (BUT4Reuse), a generic and extensible framework with feature location capabilities. Li et al. (2017) present FHistorian, a dynamic approach using software version histories.

The feature location approaches from Ziadi et al. (2012), Rubin and Chechik (2012), Xue et al. (2012), Linsbauer et al. (2013, 2014), Fischer et al. (2014), Martinez et al. (2015), and Michelon et al. (2019, 2021) are similar to our approach as they also use the commonalities and differences of product variants with regard to features. However, our approach decouples feature isolation from feature location and provides an accurate formal basis for careful preparation of a concrete feature location. This way it allows to predict and explain the possible results of a feature location, and to control and optimize the process of feature location for the available systems of a SPL. The theory presented here does not conflict with BUT4Reuse as presented by Martinez et al. (2015), rather, it is plausible that it could be successfully integrated into BUT4Reuse.

7. Conclusion

In summary, we have presented a formal theory of features in SPLE. It addresses the challenges in variability modeling by showing how feature models and software artifacts can be interrelated. Our approach uses a formal definition of feature categories, set theory, and a series of programs publicly available on GitHub that validate the theory. Our findings and contributions, including a categorization scheme for features and a formal approach for composition and decomposition of systems of a SPL provide valuable insights into this field.

The formal theory of features based on set theory is completely independent of the concrete software artifacts of the systems of a SPL. It introduces six feature categories that are equivalent to features that share parts of software artifacts, feature interactions, and aspectual features in classic feature modeling, but without considering concrete software artifacts. It describes systems as sets of features that allow composing and decomposing systems of a SPL and computing exactly which features can be isolated and subsequently located for a given number of systems of a SPL. So-called models provide various bundles of feature categories that are useful in planning and controlling feature isolation.

Theoretical validation of the theory shows that it is relatively robust with respect to missing systems, systems containing unexpected features, and features that are present in all systems of a SPL when it comes to feature isolation, which is the prerequisite for feature location.

As outlined, future work is desirable to exploit the potential of the theory to optimize the evaluation of set differences with respect to a given selection of specific systems of a SPL and to perform feature location for single systems only.

CRediT authorship contribution statement

Ulrich Eisenecker: Writing – original draft, Conceptualization, Methodology, Software, Investigation. **Richard Müller:** Writing – review & editing, Visualization, Software, Investigation, Resources, Data Curation.

Declaration of competing interest

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

Data availability

https://github.com/softvis-research/features.

Acknowledgments

We would like to express our gratitude to the anonymous reviewers. They gave us a lot of advice on how to improve the paper. In particular, they provided valuable references to related work.

References

Acher, M., Collet, P., Lahire, P., France, R.B., 2013. FAMILIAR: A domain-specific language for large scale management of feature models. Sci. Comput. Program. 78 (6), 657–681. http://dx.doi.org/10.1016/j.scico.2012.12.004, URL: https://www.sciencedirect.com/science/article/pii/S0167642312002158.

Al-Msie'deen, R., Seriai, A., Huchard, M., Urtado, C., Vauttier, S., 2013a. Mining features from the object-oriented source code of software variants by combining lexical and structural similarity. In: IEEE 14th International Conference on Information Reuse Integration (IRI). pp. 586–593. http://dx.doi.org/10.1109/IRI.2013.6642522.

Al-Msie'deen, R., Seriai, A., Huchard, M., Urtado, C., Vauttier, S., Salman, H.E., 2013b. Feature location in a collection of software product variants using formal concept analysis. In: Favaro, J., Morisio, M. (Eds.), Safe and Secure Software Reuse. Springer Berlin Heidelberg, Berlin, Heidelberg, pp. 302–307.

Apel, S., Kolesnikov, S., Siegmund, N., Kästner, C., Garvin, B., 2013. Exploring feature interactions in the wild: The new feature-interaction challenge. In: Proceedings of the 5th International Workshop on Feature-Oriented Software Development FOSD '13. Association for Computing Machinery, New York, NY, USA, pp. 1–8. http://dx.doi.org/10.1145/2528265.2528267.

Apel, S., Lengauer, C., Möller, B., Kästner, C., 2008. An algebra for features and feature composition. In: Meseguer, J., Roşu, G. (Eds.), Algebraic Methodology and Software Technology. Springer Berlin Heidelberg, Berlin, Heidelberg, pp. 36–50.

Assunção, W.K.G., Lopez-Herrejon, R.E., Linsbauer, L., Vergilio, S.R., Egyed, A., 2017. Reengineering legacy applications into software product lines: a systematic mapping. Empir. Softw. Eng. 22 (6), 2972–3016. http://dx.doi.org/10.1007/s10664-017-9499-z.

Assunção, W.K.G., Vergilio, S.R., 2014. Feature location for software product line migration: A mapping study. In: Proceedings of the 18th International Software Product Line Conference: Companion Volume for Workshops, Demonstrations and Tools - Volume 2 SPLC '14. Association for Computing Machinery, New York, NY, USA, pp. 52–59. http://dx.doi.org/10.1145/2647908.2655967.

Benavides, D., Segura, S., Ruiz-Cortés, A., 2010. Automated analysis of feature models 20 years later: A literature review. Inf. Syst. 35 (6), 615–636. http://dx.doi.org/10.1016/j.is.2010.01.001, URL: https://www.sciencedirect.com/science/article/pii/S0306437910000025.

Bigliardi, L., Lanza, M., Bacchelli, A., D'Ambros, M., Mocci, A., 2014. Quantitatively exploring non-code software artifacts. In: 14th International Conference on Quality Software. pp. 286–295. http://dx.doi.org/10.1109/QSIC.2014.31.

Cruz, D., Figueiredo, E., Martinez, J., 2019. A literature review and comparison of three feature location techniques using ArgoUML-SPL. In: Proceedings of the 13th International Workshop on Variability Modelling of Software-Intensive Systems VAMOS '19. Association for Computing Machinery, New York, NY, USA, http://dx.doi.org/10.1145/3302333.3302343.

Czarnecki, K., 1999. Generative Programming: Principles and Techniques of Software Engineering Based on Automated Configuration and Fragment-Based Component Models (Dissertation). Department of Computer Science and Automation, Technical University of Ilmenau, Ilmenau, Germany.

Czarnecki, K., Bednasch, T., Unger, P., Eisenecker, U.W., 2002. Generative programming for embedded software: An industrial experience report. In: Proceedings of the 1st ACM SIGPLAN/SIGSOFT Conference on Generative Programming and Component Engineering GPCE '02. Springer-Verlag, Berlin, Heidelberg, pp. 156–172.

Czarnecki, K., Eisenecker, U.W., 2000. Generative Programming: Methods, Tools, and Applications. Addison-Wesley, URL: http://scholar.google.com/scholar?hl=en{&}btnG=Search{&}q=intitle:Generative+Programming+Methods,+Tools+and+Applications{#}8http://scholar.google.com/scholar?hl=en{&}btnG=Search{&}q=intitle:Generative+programming:+methods,+tools,+and+applications{#}8.

Czarnecki, K., Helsen, S., Eisenecker, U.W., 2004. Staged configuration using feature models. In: SPLC. http://dx.doi.org/10.1007/b100081.

Czarnecki, K., Helsen, S., Eisenecker, U.W., 2005. Formalizing cardinality-based feature models and their specialization. Softw. Process. Improv. Pract. 10, 7–29.

El Dammagh, M., De Troyer, O., 2011. Feature modeling tools: Evaluation and lessons learned. In: Proceedings of the 30th International Conference on Advances in Conceptual Modeling: Recent Developments and New Directions ER '11. Springer-Verlag, Berlin, Heidelberg, pp. 120–129.

Fischer, S., Linsbauer, L., Lopez-Herrejon, R.E., Egyed, A., 2014. Enhancing clone-and-own with systematic reuse for developing software variants. In: IEEE International Conference on Software Maintenance and Evolution. pp. 391–400. http://dx.doi.org/10.1109/ICSME.2014.61.

Griss, M.L., Favaro, J., D'Alessandro, M., 1998. Integrating feature modeling with the RSEB. In: Proceedings. Fifth International Conference on Software Reuse (Cat. No.98TB100203). pp. 76–85. http://dx.doi.org/10.1109/ICSR.1998.685732.

IEEE Computer Society, 2014. In: Bourque, P., Fairley, R.E. (Eds.), Guide To the Software Engineering Body of Knowledge (SWEBOK), third ed. IEEE Computer Society Press, p. 346. http://dx.doi.org/10.1234/12345678, arXiv:arXiv:1210.1833v2, URL: www.swebok.org.

Kang, K.C., Cohen, S.G., Hess, J.A., Novak, W.E., Peterson, A.S., Cohen, S.G., Hess, J.A., Novak, W.E., Peterson, A.S., 1990. Feasibility Study Feature-Oriented Domain Analysis (FODA). Technical Report, Carnegie-Mellon University Software Engineering Institute, November.

Kang, K.C., Kim, S., Lee, J., Kim, K., Shin, E., Huh, M., 1998. FORM: A feature-oriented reuse method with domain-specific reference architectures. Ann. Softw. Eng. 5 (1), 143–168.

Kästner, C., Apel, S., Kuhlemann, M., 2008. Granularity in software product lines. In: Proceedings of the 30th International Conference on Software Engineering ICSE '08. Association for Computing Machinery, New York, NY, USA, pp. 311–320. http://dx.doi.org/10.1145/1368088.1368131.

Kästner, C., Dreiling, A., Ostermann, K., 2014. Variability mining: Consistent semi-automatic detection of product-line features. IEEE Trans. Softw. Eng. 40 (1), 67–82. http://dx.doi.org/10.1109/TSE.2013.45.

Kolesnikov, S., 2019. Feature Interactions in Configurable Software Systems (Phd thesis). University of Passau.

Lee, K., Kang, K.C., Lee, J., 2002. Concepts and guidelines of feature modeling for product line software engineering. In: Gacek, C. (Ed.), Software Reuse: Methods, Techniques, and Tools. Springer Berlin Heidelberg, Berlin, Heidelberg, pp. 62–77.

Li, Y., Zhu, C., Rubin, J., Chechik, M., 2017. FHistorian: Locating features in version histories. In: Proceedings of the 21st International Systems and Software Product Line Conference - Volume a SPLC '17. Association for Computing Machinery, New York, NY, USA, pp. 49–58. http://dx.doi.org/10.1145/3106195.3106216.

Linsbauer, L., Angerer, F., Grünbacher, P., Lettner, D., Prähofer, H., Lopez-Herrejon, R.E., Egyed, A., 2014. Recovering feature-to-code mappings in mixed-variability software systems. In: IEEE International Conference on Software Maintenance and Evolution. pp. 426–430. http://dx.doi.org/10.1109/ICSME.2014.67.

Linsbauer, L., Lopez-Herrejon, E.R., Egyed, A., 2013. Recovering traceability between features and code in product variants. In: Proceedings of the 17th International Software Product Line Conference SPLC '13. Association for Computing Machinery, New York, NY, USA, pp. 131–140. http://dx.doi.org/10.1145/2491627.2491630.

Martinez, J., Ordoñez, N., Tërnava, X., Ziadi, T., Aponte, J., Figueiredo, E., Valente, M.T., 2018a. Feature location benchmark with argoUML SPL. In: Proceedings of the 22nd International Systems and Software Product Line Conference - Volume 1 SPLC '18. ACM, New York, NY, USA, pp. 257–263. http://dx.doi.org/10.1145/3233027.3236402, URL: http://doi.acm.org/10.1145/3233027.3236402.

Martinez, J., Ziadi, T., Bissyandé, T.F., Klein, J., Le Traon, Y., 2015. Bottom-up adoption of software product lines: A generic and extensible approach. In: Proceedings of the 19th International Conference on Software Product Line SPLC '15. Association for Computing Machinery, New York, NY, USA, pp. 101–110. http://dx.doi.org/10.1145/2791060.2791086.

Martinez, J., Ziadi, T., Papadakis, M., Bissyandé, T.F., Klein, J., Le Traon, Y., 2016. Feature location benchmark for software families using eclipse community releases. In: Kapitsaki, G.M., de Almeida, E. (Eds.), Software Reuse: Bridging with Social-Awareness. Springer International Publishing, Cham, pp. 267–283.

Martinez, J., Ziadi, T., Papadakis, M., Bissyandé, T.F., Klein, J., le Traon, Y., 2018b. Feature location benchmark for extractive software product line adoption research using realistic and synthetic Eclipse variants. Inf. Softw. Technol. 104, 46–59. http://dx.doi.org/10.1016/j.infsof.2018.07.005, URL: http://www.sciencedirect.com/science/article/pii/S0950584918301472.

Michelon, G.K., Linsbauer, L., Assunção, W.K.G., Egyed, A., 2019. Comparison-based feature location in ArgoUML variants. In: Proceedings of the 23rd International Systems and Software Product Line Conference - Volume a SPLC '19. Association for Computing Machinery, New York, NY, USA, pp. 93–97. http://dx.doi.org/10.1145/3336294.3342360.

Michelon, G.K., Linsbauer, L., Assunção, W.K.G., Fischer, S., Egyed, A., 2021. A hybrid feature location technique for re-EngineeringSingle systems into software product lines. In: Proceedings of the 15th International Working Conference on Variability Modelling of Software-Intensive Systems VaMoS '21. Association for Computing Machinery, New York, NY, USA, http://dx.doi.org/10.1145/3442391.3442403.

Müller, R., Eisenecker, U., 2019. A graph-based feature location approach using set theory. In: 23rd Systems and Software Product Line Conference SPLC '19. ACM, Paris, France, pp. 161–165. http://dx.doi.org/10.1145/3336294.3342358.

Nie, K., Zhang, L., 2012. Software feature location based on topic models. In: 19th Asia-Pacific Software Engineering Conference, Vol. 1. pp. 547–552. http://dx.doi.org/10.1109/APSEC.2012.116.

Pohl, K., Metzger, A., 2018. Software product lines. In: Gruhn, V., Striemer, R. (Eds.), The Essence of Software Engineering. Springer International Publishing, Cham, pp. 185–201. http://dx.doi.org/10.1007/978-3-319-73897-0_11.

Raatikainen, M., Tiihonen, J., Männistö, T., 2019. Software product lines and variability modeling: A tertiary study. J. Syst. Softw. 149, 485–510. http://dx.doi.org/10.1016/j.jss.2018.12.027, URL: https://www.sciencedirect.com/science/article/pii/S016412121830284X.

Riebisch, M., Böllert, K., Streitferdt, D., Philippow, I., 2002. Extending feature diagrams with UML multiplicities. In: 6th Conference on Integrated Design & Process Technology (IDPT 2002). Pasadena, California, USA.

Rosenmüller, M., Siegmund, N., Apel, S., Saake, G., 2011. Flexible feature binding in software product lines. Autom. Softw. Eng. 18 (2), 163–197. http://dx.doi.org/10.1007/s10515-011-0080-5.

Rubin, J., Chechik, M., 2012. Locating distinguishing features using diff sets. In: Proceedings of the 27th IEEE/ACM International Conference on Automated Software Engineering. pp. 242–245. http://dx.doi.org/10.1145/2351676.2351712.

Rubin, J., Chechik, M., 2013. A survey of feature location techniques. In: Domain Engineering, Product Lines, Languages, and Conceptual Models. pp. 29–58.

Schobbens, P.-Y., Heymans, P., Trigaux, J.-C., 2006. Feature diagrams: A survey and a formal semantics. In: 14th IEEE International Requirements Engineering Conference (RE'06). pp. 139–148. http://dx.doi.org/10.1109/RE.2006.23.

Seidl, C., Winkelmann, T., Schaefer, I., 2016. A software product line of feature modeling notations and cross-tree constraint languages. In: Modellierung.

Soares, L.R., Schobbens, P.-Y., do Carmo Machado, I., de Almeida, E.S., 2018. Feature interaction in software product line engineering: A systematic mapping study. Inf. Softw. Technol. 98, 44–58. http://dx.doi.org/10.1016/j.infsof.2018.01.016, URL: https://www.sciencedirect.com/science/article/pii/S0950584917302690.

Thüm, T., Kästner, C., Erdweg, S., Siegmund, N., 2011. Abstract features in feature modeling. In: 15th International Software Product Line Conference. pp. 191–200. http://dx.doi.org/10.1109/SPLC.2011.53.

van Gurp, J., Bosch, J., Svahnberg, M., 2001. On the notion of variability in software product lines. In: Proceedings Working IEEE/IFIP Conference on Software Architecture. pp. 45–54. http://dx.doi.org/10.1109/WICSA.2001.948406.

Wahyudianto, Budiardjo, E.K., Zamzami, E.M., 2014. Feature modeling and variability modeling syntactic notation comparison and mapping. J. Comput. Commun. 2, 101–108. http://dx.doi.org/10.4236/jcc.2014.22018.

Xing, Z., Xue, Y., Jarzabek, S., 2013. A large scale Linux-Kernel based benchmark for feature location research. In: 35th International Conference on Software Engineering (ICSE). pp. 1311–1314. http://dx.doi.org/10.1109/ICSE.2013.6606705.

Xue, Y., Xing, Z., Jarzabek, S., 2012. Feature location in a collection of product variants. In: 19th Working Conference on Reverse Engineering. pp. 145–154. http://dx.doi.org/10.1109/WCRE.2012.24.

Zaatar, M.A., Hamza, H.S., 2011. An approach for identifying and implementing aspectual features in software product lines. In: 23rd International Conference on Software Engineering & Knowledge Engineering.

Ziadi, T., Frias, L., da Silva, M.A.A., Ziane, M., 2012. Feature identification from the source code of product variants. In: 16th European Conference on Software Maintenance and Reengineering. pp. 417–422. http://dx.doi.org/10.1109/CSMR.2012.52.

Dr. Ulrich W. Eisenecker is Professor of Information Systems, in particular Software Development for Business and Administration at the University of Leipzig. His research focuses on Software Product Lines, Software Visualization in 3D, and E-Assessments. Together with Krzysztof Czarnecki, he is the author of the classic book "Generative Programming. Methods, Tools, and Applications" which was published in 2000.

Dr. Richard Müller is Senior Expert at Deloitte. He has over 14 years of professional experience in the areas of Requirements Engineering, Software Architecture, Software Engineering, Agile Project Management, and IT security. He has gained this experience both in practice and in academia as a software developer/architect, product owner, data protection officer, research group leader, and lecturer. His research covers Software Engineering, Software Architecture, Software Analytics/Mining Software Repositories, Software Visualization, and Software Product Lines.