An Overview of Feature-Oriented Software Development

Sven Apel, Department of Informatics and Mathematics, University of Passau, Germany
Christian Kästner, School of Computer Science, University of Magdeburg, Germany

Feature-oriented software development (FOSD) is a paradigm for the construction, customization, and synthesis of large-scale software systems. In this survey, we give an overview and a personal perspective on the roots of FOSD, connections to other software development paradigms, and recent developments in this field. Our aim is to point to connections between different lines of research and to identify open issues.

1 INTRODUCTION

Feature-oriented software development (FOSD) is a paradigm for the construction, customization, and synthesis of large-scale software systems. The concept of a feature is at the heart of FOSD. A feature is a unit of functionality of a software system that satisfies a requirement, represents a design decision, and provides a potential configuration option. The basic idea of FOSD is to decompose a software system in terms of the features it provides. The goal of the decomposition is to construct well-structured software that can be tailored to the needs of the user and the application scenario. Typically, from a set of features, many different software systems can be generated that share common features and differ in other features. The set of software systems generated from a set of features is also called a software product line [45,101].

FOSD aims essentially at three properties: structure, reuse, and variation. Developers use the concept of a feature to structure design and code of a software system, features are the primary units of reuse in FOSD, and the variants of a software system vary in the features they provide. FOSD shares goals with other software development paradigms, such as stepwise and incremental software development [128,98,99,105], aspect-oriented software development [58], component-based software engineering [65,118], and alternative flavors of software product line engineering [45,101], whose differences to FOSD we will discuss.

Historically, FOSD has emerged from different lines of research in programming languages, software architecture, and modeling. So it is not surprising that current developments in FOSD comprise concepts, methods, language, tools, formalisms, and theories from many different fields. An aim of this article is to provide a historical overview of FOSD as well as a survey of current developments that have...
emerged from the different lines of FOSD research. Due to the sheer volume and diversity of work on FOSD and related fields, we cannot hope for completeness. Instead, we provide a personal view on recent interesting developments in FOSD. Also we do not aim at a comprehensive discussion of individual approaches but at highlighting connections between different lines of FOSD research and identifying open issues.

2 CONCEPTS AND TERMINOLOGY

2.1 What is a Feature?

FOSD is not a single development method or technique, but a conglomeration of different ideas, methods, tools, languages, formalisms, and theories. What connects all these developments is the concept of a feature. Due to the diversity of FOSD research, there are several definitions of a feature \[42\], e.g. (ordered from abstract to technical):

1. Kang et al. \[70\]: “a prominent or distinctive user-visible aspect, quality, or characteristic of a software system or systems”

2. Kang et al. \[71\]: “a distinctively identifiable functional abstraction that must be implemented, tested, delivered, and maintained”

3. Czarnecki and Eisenecker \[47\]: “a distinguishable characteristic of a concept (e.g., system, component, and so on) that is relevant to some stakeholder of the concept”

4. Bosh \[35\]: “a logical unit of behaviour specified by a set of functional and non-functional requirements”

5. Chen et al. \[40\]: “a product characteristic from user or customer views, which essentially consists of a cohesive set of individual requirements”

6. Batory et al. \[30\]: “a product characteristic that is used in distinguishing programs within a family of related programs”

7. Classen et al. \[42\]: “a triplet, \( f = (R, W, S) \), where \( R \) represents the requirements the feature satisfies, \( W \) the assumptions the feature takes about its environment and \( S \) its specification”

8. Zave \[129\]: “an optional or incremental unit of functionality”

9. Batory \[24\]: “an increment of program functionality”

10. Apel et al. \[21\]: “a structure that extends and modifies the structure of a given program in order to satisfy a stakeholder’s requirement, to implement and encapsulate a design decision, and to offer a configuration option”
From top to bottom, the definitions become less abstract and more technical. While the first seven definitions reflect that features are abstract concepts of the target domain, used to specify and distinguish software systems, the last three definitions capture the fact that features must be implemented in order to satisfy requirements. This tension between abstract and implementation view is not accidental. It stems from the different uses of features. For example, in feature modeling, features are used to describe the variability of a software product line for communication with stakeholders, independent of any implementation, as we will discuss in Section 4.1. In feature-oriented programming, a feature is a first-class language construct for structuring source code, as we will discuss in Section 4.3.

Czarnecki’s and Eisenecker’s distinction between problem space and solution space, as illustrated in Figure 1, is useful for the classification of the definitions presented above [47]. The problem space comprises concepts that describe the requirements on a software system and its intended behavior. The solution space comprises concepts that define how the requirements are satisfied and how the intended behavior is implemented. The first seven definitions describe the feature concept from the perspective of the problem space. Here, features are used to describe what is expected from a software system. The last three definitions describe features from the perspective of the solution space, i.e., how features provide/implement the desired functionality.

2.2 What is Feature-Oriented Software Development?

The concept of a feature is useful for the description of commonalities and variabilities in the analysis, design, and implementation of software systems. FOSD is a paradigm that favors the systematic application of the feature concept in all phases of the software life cycle. Features are used as first-class entities to analyze, design, implement, customize, debug, or evolve a software system. That is, features not only emerge from the structure and behavior of a software system, e.g., in the form of the software’s observable behavior, but are also used explicitly and systematically to
define variabilities and commonalities, to facilitate reuse, and to structure software along these variabilities and commonalities. A distinguishing property of FOSD is that it aims at a clean (ideally one-to-one) mapping between the representations of features across all phases of the software life cycle. That is, features specified during the analysis phase can be traced through design and implementation.

The idea of FOSD was not proposed as such in the first place but emerged from the different uses of features. A main goal of this survey is to convey the idea of FOSD as a general development paradigm. The essence of FOSD can be summarized as follows: on the basis of the feature concept, FOSD facilitates the structure, reuse, and variation of software in a systematic and uniform way.

2.3 Phases of the FOSD Process

The FOSD process comprises four phases: (1) domain analysis, (2) domain design and specification, (3) domain implementation, and (4) product configuration and generation. As stated before, a distinguishing property of FOSD is that the feature concept pervades all of these phases. Domain analysis determines which features are part of the software system or software product line. This includes also a domain scoping step that defines the dimension of the domain, i.e., which features are supported and not supported. For example, a company may find it more effective to concentrate on a product line of database systems for automotive systems instead of databases in general. Furthermore, developers gather relevant knowledge about how features relate (e.g., which features require the presence or absence of other features; see Section 4.1). For example, in a database product line, the feature for transaction management is selectable only if the database supports writing access since in a read-only database no transactions are needed [109,108]. In the design and implementation phases, the developers typically create a set of first-class feature artifacts that encapsulate design and implementation information (see Sections 4.2 and 4.3). In our database example there would be a modular design and implementation of a basic database system and modular designs and implementations of features like transaction management, storage management, and query processing. Based on the domain knowledge and the feature artifacts, applications can be generated almost automatically. Due to the clean mapping between features and feature artifacts, the user only needs to specify the desired features by name (see Section 4.4) and, based on the domain knowledge, a generator can pick the corresponding feature artifacts and create an efficient and reliable software product (see Section 4.4). In the database scenario, a generator could, for example, automatically select a B-tree index in favor of a hash map because, in the case of large amounts of data, the B-tree grants faster access.

In Figure 2, we illustrate the phases of the FOSD process. For each phase, we show some key technologies, which are explained in the next sections. Rather than explaining them here, the figure is designed to be a reference and signpost for the remaining paper. Note that theory is not a distinct phase (box at the bottom of.
2 CONCEPTS AND TERMINOLOGY

Figure 2), but provides a foundation for all phases. One may say that the process model is too simplistic for real-world circumstances. This is true, but its purpose is to structure our survey and not to define the development process in a company.

2.4 Why is FOSD Special?

There is a substantial overlap of FOSD with other software development paradigms. We explain key differences to some popular alternatives.

The idea of stepwise and incremental software development (SISD) is to encapsulate individual development steps that implement distinct design decisions [128, 98, 99, 105]. The result is usually a total (or at least a partial) order of encapsulated development steps (a.k.a. refinements or layers [98, 30]). The goals of this approach are to structure software in order to support change (by removing and adding layers). FOSD shares these goals (and has further goals such as reuse and variation). However, many ideas and concepts of FOSD emerged from SISD. In fact, FOSD is the forcefully broadened development of these early ideas in the context of large-scale software synthesis and software product lines.

Aspect-oriented software development (AOSD) aims at the modularization of crosscutting concerns. A crosscutting concern is a concern that does not align well with the structure established by object-oriented or functional decomposition [81, 119]. It has been observed that features are frequently crosscutting concerns [116, 91, 18, 74], so the implementation of features can benefit from aspect-oriented techniques [96, 18]. While, initially, variation and reuse have not been the main goals of AOSD, there have been considerable efforts in this direction [61, 90, 87]. We believe that, at some point, both paradigms became difficult to distinguish at the implementation level but FOSD’s aim at simplicity [6, 18], automated software synthesis (see Section 4.4), and algebraic models (see Section 4.5) are still distinguishing characteristics.

Component-based software engineering (CBSE) aims at the vision of constructing software systems on demand using off-the-shelf components [65, 118]. Work on service-oriented architectures is a modern instance of this vision [57]. A main difference to FOSD is that components/services are typically black boxes. While this facilitates independent development and deployment, it hinders encapsulating crosscutting features of a software system [100, 14]. For example, detaching the code of the transaction management from a database system and encapsulating it in a dedicated component is not possible in practice [74, 109]. It has been observed that features are often program slices, i.e., crosscutting concerns [5]. That is, features often do not align well with the decomposition imposed by component models. Using components anyway leads to a non-trivial mapping between features and components [60, 34, 4, 1].

The paradigms of software product line and domain engineering aim at families

1A recent branch of this paradigm is change-oriented software engineering [56].
Figure 2: Phases of the FOSD process.
3 THE ROOTS OF FOSD

There are three major lines of research that have contributed to the state of the art of FOSD: feature modeling, feature interaction, and feature implementation.

3.1 Feature Modeling

In their seminal work on feature-oriented domain analysis (FODA), Kang et al. were the first to introduce and use the concept of a feature for the description of the commonalities and variabilities of software systems [70]. Their aim was to structure the problem space (cf. Figure 1). They introduced the notion of a feature model describing the relationships and dependencies of a set of features belonging to a particular domain. In Figure 3, we show a standard example of the feature modeling community in a commonly used feature diagram notation [47]: a feature model that describes the variability of a simple car, i.e., the variants of cars that can be produced in this domain. The root of the tree denotes the concept that is modeled. All other boxes denote features, where child features depend on parent features. There are different types of model elements that describe constraints on the way in which features can be combined, as shown in Figure 3. For example, every car has a body, transmission, and engine (filled circle), but a car does not necessarily have a trailer (empty circle) or the engine may be powered with gasoline or with electricity or with both (filled arc).

In contrast to FOSD, the early work on FODA did not make features explicit in design and code, which led to a complex mapping between features and design and implementation artifacts. Nevertheless, feature models have been used successfully in many academic and industrial projects and drive the research and best practice in software product line engineering [70, 47, 86, 121]. They are the de facto standard
in modeling variability in product line engineering, and further work on FOSD will build on this success. Since the introduction of feature models, several refinements and extensions have been proposed, some of which we will discuss in Section 4.1.

3.2 Feature Interaction

In a different line, researchers and practitioners – mainly from the telecommunications industry – have developed the notion of feature interaction [36,39]. Although largely developed independently of the work of Kang et al., they use features in a similar sense but with focus on their run time interaction. A feature interaction is a situation in which two or more features exhibit unexpected behavior that does not occur when the features are used in isolation. The standard example is a phone with – beside basic functionality – two features call waiting and call forwarding: when used in isolation, both features work fine, but when used in combination, it is unclear what to do with an incoming call on a busy line. The call is either forwarded or announced; in either case, the expected behavior of one of the two features is compromised.

Let us illustrate the problem of feature interaction further by means of a simple code example, which we learned about from Wolfgang Scholz in personal communication. Suppose we have a straightforward implementation of a singly linked list, as shown in Figure 4 (feature SINGLE). Further suppose an implementation of a reversely linked list, as shown in Figure 5, which is also implemented straightforwardly (feature REVERSE). Both features, SINGLE and REVERSE, work without problems as long as they are isolated. But having both features in a single implementation, henceforth called a doubly linked list, leads to an unexpected behavior. In Figure 6, we show the list implementation including both features. In this case, the code of
the two features is simply merged (i.e., a structural union); in other examples, the merging is more difficult. Adding an element to the end or in front of the doubly linked list does not preserve the connectivity between the list elements. The reason is that, when adding an element in front of the list with method \texttt{push}, the backward reference \texttt{prev} is not adjusted accordingly and the reference to the last element is not set properly in the case the list is empty – a similar problem occurs in the opposite case. In Figure 7, we show the implementation containing the two features – but now we have fixed the problem by adjusting the references properly. Note that the fixes (underlined) are effective only when both features are present. So the additional code is neither part of feature \texttt{Single} nor of feature \texttt{Reverse} when used in isolation.

Feature interaction, as illustrated above, poses a major problem in the development of feature-oriented software systems \cite{36,39,89,88,82,79}. There are several techniques for detecting and handling interactions that address this issue (see Section 4.3).

### 3.3 Feature Implementation

Prehofer was the first who recognized the need for making features (and feature interactions) explicit at the programming language level \cite{104}. He proposed to treat the additional code elements, which a feature applies to a base program, as first-class entities and to separate them from the base code. This allows a developer to trace features easily from the problem space to the solution space, which had been a longstanding problem \cite{60}. The problem of separating the code of different features follows the seminal principle of separation of concerns, credited to Dijkstra \cite{54} and Parnas \cite{98,99}.

Furthermore, Prehofer explored the feature interaction problem at the level of the static program structure. He proposed to factor out code that is necessary to control the interaction of two features in a separate module, called a lifter \cite{104}. In
our linked list example, a lifter would contain the code for adjusting the references to the next and previous elements (underlined in Figure 7).

Technically, Prehofer’s approach builds on the concept of collaboration-based design [106, 126, 116] and mixin techniques [38, 59] to separate feature-related code from the base program. There is a wide variety of work on components and software composition that predates or complements Prehofer’s work and that pursued related ideas, e.g., GenVoca [29], subject-oriented programming [63], aspect-oriented programming [81], adaptive plug-and-play components [95], and role components [126]. All aim at the modularization of software at a greater scale than functions and classes. Although not intended for FOSD, these approaches can be used to implement features cohesively in order to be able to compose them in different combinations, as illustrated in Figure 8. Interestingly, Batory’s work on GenVoca led to several FOSD languages, tools, and theories, which we discuss in the next sections.

3.4 Discussion

Although the three roots of FOSD appear different at first glance, they all share the concept of a feature to describe and implement commonalities and variabilities of software systems. Only in recent work the different lines begin to converge toward a unified development methodology, as illustrated in Figure 9. An aim of this survey is to guide and accelerate this process and to strengthen the identity of the FOSD community. In the following sections, we bring together some influential and promising work in the field of FOSD.

4 RECENT DEVELOPMENTS IN FOSD

We structure our discussion of recent work on FOSD along the distinct phases of the FOSD process. We begin with domain analysis and proceed over domain design and implementation toward product configuration and generation. An exception is our discussion of FOSD theory at the end, which “cuts across” all phases of FOSD.
4.1 Domain Analysis

Feature modeling is the central activity in domain analysis. The aim is to identify and capture the variabilities and commonalities of a domain, in the case of FOSD, in the form of a feature model. FODA was the first analysis method that came with its own kind of feature model, as shown in Figure 3. Several analysis methods and processes emerged from FODA, e.g., FORM [71] and FeatureRSEB [62]. These mainly refine and extend the domain analysis process, but features remain the central concept of both approaches. Schobbens et al. compare different kinds of feature models and provide a unifying formal semantics [111].

Recently, on the basis of FODA, several extensions of feature modeling and feature models were proposed. A major line of research aimed at enriching feature models with additional information, e.g., regarding feature cardinalities [48], propositional constraints [24], and non-functional properties of features [113]. The idea is to use this information in the product configuration and generation process in order to rule out invalid or suboptimal product variants (see Section 4.4). Nevertheless, there is a trade-off between expressiveness and simplicity of feature models. The more information is exposed in the model, the more it can guide the configuration and generation process, but the more complex the model becomes. That is, more expressive models are often not “management compatible”, which was their initial strength. In further work, this trade-off should be explored in depth.

As feature models became larger, the necessity of automated reasoning became apparent [33, 28]. Mannion [92] and Batory [24] were among the first to note the connection between feature models and propositional formulas. A translation of a feature model into a proposition enables tools to reason about properties of the feature model automatically. For example, tools can answer questions such as: Does a feature selection satisfy constraints imposed by a feature model [24]? How many variants can be generated [32]2? Are two feature models identical [122]? Which features and relationships have to be changed to fix an inconsistent feature model [127]? Furthermore, several researchers noted the relationship to grammars and design wizards.
(i.e., tools that automate the selection of features in order to optimize non-functional properties) and presented approaches to generate, based on feature models, design wizards that present and constrain the configuration options [52, 3, 34, 24, 80]. A subproblem of large feature models is that automated reasoning is time consuming since, in many cases, computing interesting properties is \( \mathcal{NP} \)-complete (e.g., the satisfiability and equivalence of feature models). Hence, several researchers have explored how to reason about feature models efficiently. For example, binary decision diagrams [94], satisfiability solvers [93], constraint satisfaction solvers [32], or simplified reasoning [122] were used to reason about feature models of up to 10,000 features. In order to guarantee the correctness of analyses and manipulations performed on large-scale feature models, Trinidad et al. propose a framework for the automated error detection [123].

Like any other kind of software artifact, feature models evolve. Hence, programmers need support for refactoring and reverse engineering. The biggest issue is to guarantee that changes to a feature model do not produce errors. Recently, some researchers addressed this issue by providing proper tool support based on efficient reasoning approaches (see above) [51, 50, 122].

To summarize, feature modeling has received much attention by researchers and practitioners in recent years. A trend is to extend and use feature models for automatic product generation, but this complicates the use of features for users who are not domain experts (e.g., managers or customers). While this trend has to be followed in order to realize the full potential of FOSD, it is also important to explore the trade-off between expressiveness and simplicity.

### 4.2 Domain Design and Specification

Domain design and specification is the process of defining the architecture of a software product line. In the context of FOSD, this means that the essential structural and behavioral properties of the involved features are specified using a formal or informal specification and/or modeling language. Remarkably, there has not been much work in this direction. FOSD researchers concentrated mainly on feature modeling and feature implementation. That is, once the features and their relationships have been set in a feature model, the features are implemented. This is in stark contrast to non-FOSD approaches, in which developers have to design a product line architecture first in order to define the granularity of components or extension points in a common framework [101]. In FOSD, features structure the design of the software system. Hence, modeling and specification activities do not aim at defining a variable architecture (this is given by the decomposition into features) but at defining the structure and behavior of features and their interactions. Modeling and specification languages have to take features and their potential for combinations and interactions into account.

First attempts have been made to specify and model the structure and behavior of a feature in isolation and to ensure the safe composition of features without con-
sidering the implementation. Poppelton proposed an approach for feature-oriented specification [102]. He extends Event-B, a widely used formal specification language for safety critical systems, with support for feature composition. The (yet unreached) goal is to propagate proven feature correctness properties through feature composition to the generated software product.

Recent work of Trujillo et al. aims at connecting FOSD with model-driven development (MDD) [124]. The key idea is to unify features with MDD transformations. Features are vertical transformations (which elaborate one representation); MDD transformations are horizontal transformations (which map one representation to a different representation). Both are functions, but they are used in different circumstances. Batory et al. proposed a foundation based on category theory to describe the interplay of and constraints on features and MDD transformations [25, 26, 27]. In Batory’s work it is not considered how a feature itself is modeled or specified. Diskin et al. used a similar model to solve some problems in model versioning [55], in which features correspond to updates applied to models.

Furthermore, Apel et al. explored how features can be modeled using standard modeling techniques such as the *unified modeling language (UML)* [8]. The idea is not to model the individual programs that can be derived from a product line but the individual features (e.g., using class, state, and sequence diagrams) and so on, and to merge the models in a prescribed way when a product is generated. Left of the equality in Figure 10, we show the class diagrams of the two features SINGLE and REVERSE of our list example (‘●’ denotes feature composition). Right of the equality, we show the merged model that contains the two features and that describes a doubly linked list. The (de)composition of models into features is related to other fields such as aspect-oriented modeling [41, 83, 68, 67].

An alternative approach to represent features in modeling is not to decompose a model along features but to annotate a given model with information on features [46, 64, 112]. This way, we do not have multiple model fragments that correspond to individual features but a single, superimposed model which contains annotations pointing to features.

Finally, there is a significant body of work on specifying and resolving feature interactions at the design/specification level [39, 107]. The problem is that, especially
in distributed systems, the accidental interaction of features becomes only apparent at run time in the form of misbehavior. In order to detect such misbehavior, it is necessary to specify/model the intended behavior. Since the early work on feature interaction, much progress has been made in detecting and resolving feature interactions, for example, using static analysis [69] and model checking [85]. Most of this work aims at networks and distributed systems and is outside the scope of this paper. The connection to other subfields of FOSD, such as feature modeling and implementation, has not been drawn yet. Some work on specification and verification addresses also the feature interaction problem. We discuss this work in Section 4.4 together with other approaches that aim at correct product generation.

To summarize, there are many open issues in modeling and specifying features. For example: How can we verify that a composition of features satisfies the specification of the individual features and the desired product? We hope that this survey will encourage more work in this direction.

4.3 Domain Implementation

As stated before, Prehofer was the first to note the necessity of making features explicit in code [104]. The goal is to establish a one-to-one mapping between features that appear during the domain analysis and features that appear at the implementation level. Prehofer proposed first-class language constructs to represent the changes and additions a feature makes when being added to a program. Since then, much progress has been made in this direction.

There are several languages that support the concept of features explicitly. They all share the goal to provide better abstraction and modularization mechanisms for features. For example, Jak is a language that extends Java by feature-oriented mechanisms [30]. Code belonging to a particular feature is stored in a dedicated directory, called a containment hierarchy [30]. Typically, a containment hierarchy contains multiple class and refinement declarations. A refinement declaration is a way to apply the changes a feature makes to a program subsequently, without changing the program’s code. In Figures 11 and 12, we show our list example implemented in Jak. In Figure 11, we have the two classes List and Node of feature SINGLE. These classes are identical to the ones implemented in plain Java (cf. Figure 4), except the layer declaration, which specifies the enclosing feature. In Figure 12, there are the two refinement declarations of feature REVERSE that add fields and methods to the classes of feature SINGLE. The result of composing the features SINGLE and REVERSE behaves like the code shown in Figure 6.

Beside Jak, also feature-oriented language extensions were proposed for C++, called FeatureC++ [17], and for XML, called Xak [2]. Basically, feature-oriented languages rely on programming mechanisms such as mixins [38, 59] and collaborations [106, 116, 126], but also aspects and subjects became popular in FOSD [61, 96, 87, 90, 74, 12, 18]. However, the connection of the original mechanisms to the other phases of the FOSD process is typically not made explicit.
An interesting recent trend is to explore the principles of feature modularity independently of a particular language [30, 91, 7, 15, 37]. The AHEAD tool suite was the first to take advantage of a language-independent model of features (see Section 4.5) and provides support for implementing features in different languages, e.g., Java and grammar specifications [30]. However, the integration of new languages into the AHEAD tools suite was ad hoc and tedious. The FeatureHouse tool suite follows the lead of AHEAD and provides an easy-to-use plug-in mechanism for new languages, based on attribute grammars [15]. This way, many very different languages like C, Haskell, JavaCC, or XHTML could be prepared for implementing and composing features [15, 8, 10].

The work on implementing features has been influenced by the work on feature interactions. Prehofer was the first to note the relevance of feature interactions for implementing features [104]. He observed that, in the presence of feature interactions, the implementation of a feature may vary, as in the example of the doubly linked list (see Section 3.3). Prehofer proposed to separate interaction code from basic feature code in dedicated modules called lifters, i.e., to implement multiple modules per feature, which is a slight departure from the initial goal of work on FOSD (namely, a one-to-many instead of a one-to-one mapping between features and feature artifacts). Liu et al. extended the notion of lifters to derivatives in order to capture higher-order interactions (i.e., interactions between pieces of interaction code) and presented a theory for feature interactions [88] (see Section 4.5). In Figure 13, we show the lifter/derivative SINGLEREVERSE that fixes the interaction between the features SINGLE and REVERSE by overriding the methods push and shup and adding code for adjusting the references properly (Super(Node) refers to the method that is been refined.). SINGLEREVERSE is only present when both SINGLE and REVERSE are present.

Implementing features in separate feature artifacts (e.g., flavors of modules, containers, or components) is not the only way to make features explicit in the implementation. A view on feature implementation that differs from feature-oriented programming emerged from simple variant and configuration mechanisms like the C preprocessor. The idea is not to implement features using dedicated abstraction and modularization mechanisms, but to annotate code with information on fea-
features [72,16], which is basically like annotating models with feature information, as discussed in Section 4.2. For example, in the code base of a database system, one could annotate all code lines with #ifdef-like directives that implement transaction safety. This way, transaction management can be switched on and off at compile time using a single flag. That is, two variants of the database system can be generated, one with and one without transaction management. Additionally, a simple search mechanism can be used to find all code belonging to the transaction feature. Annotation mechanisms such as the C preprocessor are used widely in industry to implement features. An advantage is the fine-grained model of extensions (even statements and expressions can be annotated) [75] and the easy adoption [44]. A disadvantage is that features are not encapsulated in cohesive units but scattered across the code base [72,16]. There seems to be a trade-off between modularity and expressiveness or ease of use that is an interesting open research issue.

Commercial product line tools like Gears [84] or pure::variants [34] also support the annotation of code with features. Furthermore, both connect feature models to annotations to achieve a mapping between features in the problem and solution space, but the mapping is usually complex and annotations obfuscate the source code, which results in code that is criticized as “#ifdef hell” [117]. Recently, advances have been made to address these problems [75]. The idea is to provide first-class feature support in an integrated development environment, implemented in a tool called Colored IDE (CIDE), and to separate annotations from code in order not to pollute the code base. Annotations are managed externally and displayed using background colors. A principle of “disciplined” annotations limits annotations to meaningful code fragments [75] and, like in FeatureHouse, CIDE has been designed extensible in order to be able to integrate new languages rapidly [78,77]. In Figure 14, we show the implementation of the list example using CIDE. Code belonging to the features SINGLE and REVERSE has been colored. Interestingly, the problem of varying feature implementations in the presence of feature interactions occurs also in the context of annotation-based approaches [82]. In particular, nested annotations indicate interactions (see Figure 14). The inner-most nest corresponds to a lifter. In CIDE, feature interactions become apparent when colors denoting different feature overlap (the colors are blended in the overlapping region, cf. Figure 7).
To summarize, feature implementation has been receiving much attention in recent years, by programming language designers and tool developers. An open issue is how to combine annotation-based and composition-based approaches in a unified and efficient framework. Furthermore, the idea of factoring feature interaction code via lifters and derivatives is promising, but it is not clear whether this approach scales to a large number of features with many interactions.

4.4 Product Configuration and Generation

In FOSD, generation plays a key role. The vision is that, following a user’s feature selection, an efficient software system is generated automatically, as illustrated in Figure 15. This differs from the traditional approach of domain and application engineering, in which the user is in charge of assembling, adapting, and integrating the reusable assets produced by domain engineering. Full automation is one of the key goals of FOSD.

Several steps are needed to generate an efficient software system from a user’s feature selection. First, to assist the user in selecting a set of desired features, tools need to present the available features as well as their constraints and relationships clearly. Ideally, invalid feature selections are reported and rejected interactively, i.e.,
Figure 15: Automatic product generation on the basis of a feature selection [30].

during the feature selection process. Tools like GUIDSL [24], pure::variants [34], FeatureIDE [80], CIDE [76], and many others present available features in the form of interactive feature models (represented as radio buttons and check boxes or trees) and assist users in finding a valid selection by evaluating the relationships in the background, visualizing relationships, and hiding invalid options. The next step is to compute a complete, valid feature selection on the basis of a partial feature selection by evaluating possible complete feature combinations and judging their appropriateness. The background is that real-world product lines have many hundreds or even thousands of features, whose sheer number certainly overburden the average user. The simplest approach would be to present all feature selections to the user that are possible. More appropriate would be to use domain knowledge in order to determine which features work well with other features. In the latter scenario, the word 'well' is the key. What does it mean for two features to work well? Clearly more information than just the feature model is needed. Non-functional properties of features, provided by the user and/or computed by a tool, can guide the feature selection process, as proposed by Siegmund et al. [114] and Sincero et al. [115]. For example, profiling reveals that certain index structures (e.g., B-tree, T-tree, hash map) of a database system obey different footprint and performance characteristics in different situations (e.g., amount of data, access patterns, query types). This knowledge can be used during feature selection to optimize the generated database system either toward maximal performance or minimal footprint. Another strategy for automatically selecting the 'best' features is to analyze the context in which the variant is to be used [110].

The process of optimizing a feature selection is a form of architectural metaprogramming. Architectural metaprogramming (a.k.a. computational design) is a general approach to software development that provides a unifying view on FOSD, MDD, and refactoring [26, 27]. Architectural metaprogramming can be explained by contrasting it to programming in the small. The basic idea is that programming in the small is creating objects and writing methods that update these objects or that translate objects of one type to another. Programming in the large is where ob-
jects represent system designs, and methods are transformations that update these designs or translate designs in one representation to that of another. In this view, a feature selection represents a system design or architecture and optimization is a transformation that updates or changes the design or architecture.

Once we have a proper feature selection, the desired software system is generated. Variant generation (a.k.a. product derivation) concerns all kinds of artifacts, from code over models to documentation. As stated in Section 4.3, the work on AHEAD was the first that aimed at an integrated FOSD tool for software artifacts written in different languages. The next generation are tools like FeatureHouse and CIDE, which automate also the integration of new languages to build language-independent composition or generation tools.

An important question during generation is whether the generated code is correct. There are three levels of correctness: (1) syntactic correctness, i.e., the generated software system is correct with respect to the language’s syntax, (2) type correctness, i.e., the generated software system is well-typed with respect to the language’s type system, and (3) behavioral correctness, i.e., the generated software system behaves correctly according to a formal or informal specification. Furthermore, it is desirable to guarantee correctness properties for the entire product line instead of checking each product variant in isolation. The reason is that generating all systems is not feasible due to the potentially large number of valid feature selections (already for 33 independent, optional features, a variant can be generated for every person on the planet).

With regard to syntactic and type correctness of product lines, researchers made considerable progress in recent years [49, 120, 73, 13, 53, 78, 11]. Most prominently, Pietroszek and Czarnecki [49] and Thaker et al. [120] implemented type systems for UML- and AHEAD-based product lines. Also, attempts have been made to formalize type systems for feature-oriented product lines [73, 53, 11]. In contrast, behavioral correctness is difficult to guarantee, due to the lack of feature support of contemporary specification, verification, and validation techniques, but first steps have been made. For example, configuration lifting is a technique that translates feature information into metaprograms and uses established verification techniques to ensure correctness [103]. Classen et al. propose to merge the transition systems representing the behaviors of different features into a single, superimposed transition system and to use information on features and model checking techniques for efficient verification [43]. Alternative approaches, that propose to generate proofs, are promising but have not been considered with regard to software product lines [23].

Another approach of attaining confidence in the generation process is to test or validate a product line [101]. While tests cannot guarantee the absence of errors, they can provide a certain degree of confidence. However, as with type checking, testing all variants of a software product line is in the most cases not feasible. Instead, meaningful subsets [101], model-based and composable tests [97, 125] have

---

Note that we could have discussed this issue also in Section 4.3, as it is at the fringe between implementation and generation.
to be used. The problem of feature interaction is important in this context since features tested in isolation may obey the desired behavior but their combination may not, as the list example has taught us. It is an open issue whether the entire code base of a product line can be tested instead of generating and testing products individually.

To summarize, in recent years techniques and tools from the fields of feature modeling and feature implementation begin to converge toward a unified framework for FOSD. The main challenges are how to specify, verify, and test the correctness of software products generated from product lines, ideally without generating all products. Further important issues are how to represent domain knowledge and how to generate efficient software products based on this information.

4.5 Theory

The first theory on FOSD, even though at the time it was not called FOSD, was GenVoca. GenVoca models the changes that a feature applies as function application and feature composition as function composition, denoted by \( \bullet \). For example, a product line of telecommunication systems consisting of the three features PHONE, CALLWAITING, and CALLFORWARDING is modeled as a set:

\[
\{ \text{PHONE, CALLWAITING, CALLFORWARDING} \}
\]

Different variants of telecommunication systems are modeled by composing functions (that represent features) in different combinations:

\[
\begin{align*}
\text{PHONE} \\
\text{CALLWAITING} \bullet \text{PHONE} \\
\text{CALLFORWARDING} \bullet \text{PHONE} \\
\text{CALLWAITING} \bullet \text{CALLFORWARDING} \bullet \text{PHONE} \\
\ldots
\end{align*}
\]

Currently, it is not clear whether the order of features plays or should play a role in modeling product lines. In the case of the list example, the composition order does not matter. In other cases, this may be different [9]. Generally, it has been shown that the order of features depends on the implementation mechanism [15] and that it can be changed by refactorings [22, 82, 9]. But should the user be aware of the order, or should the user just pick the features she needs?

AHEAD is the successor of GenVoca. It aims at language independence in that it represents the changes that a feature applies as nested records, in which each element denotes a certain type of artifact. For example, the record below represents feature SINGLE of the list example (including two Java classes and a documentation in HTML):

\[
\text{SINGLE} : \{ \text{Java} : \{ \text{List} : \{ \text{first, push} \}, \text{Node} \{ \text{next} \} \}, \text{HTML} : \{ \text{List.html} \} \}
\]
Another approach is to use specific trees, called *feature structure trees* [19, 15], as shown in Figure 16.

Both approaches (nested records and feature structure trees) share many similarities but also differ in some respects. This is reflected in the fact that two different algebras have been developed [31, 21]. In both algebras, features are represented by algebraic expressions and feature composition is performed by dedicated operations. For example, in the feature algebra of Apel et al. [21, 20], feature SINGLE is represented as follows:

\[
\text{SINGLE} ::= \text{Node}.next : \text{Field} \oplus \text{Node} : \text{Class} \oplus \text{List}.push : \text{Method} \oplus \text{List}.first : \text{Field} \oplus \text{List} : \text{Class}
\]

Feature REVERSE is represented analogously. The composition of the two features is the addition (using operator \(\oplus\)) of the summands of the two features’ algebraic expressions.

The initial idea of using algebra for describing and comparing programming techniques is due to Lopez-Herrejon et al. [91]. A benefit of algebraic models is that they provide insight into the fundamental properties of feature composition, e.g., why feature composition is usually *not* commutative. Furthermore, algebra has been used to model and reason about feature interactions [89, 88] and product line variability [66]. Finally, in order to realize the vision of architectural metaprogramming, formal models like the feature algebra are essential [26].

Trujillo et al. extended the AHEAD theory toward model-driven development, called *feature-oriented model-driven development (FOMDD)* [124]. The idea is to integrate software artifacts that reside on different abstraction levels (levels in the model stack) and to model transformations of them uniformly. Beside features, also model refinement involves a transformation. FOMDD reveals their relationship, as illustrated in Figure 17.

Still an open issue is how to connect theories at different levels, e.g., at the feature model level [66], the design/modeling level [124], or the implementation level [31, 21], into a consistent unified theory that describes all structures and mechanisms used in the FOSD process.
5 PERSPECTIVE

Since the concept of a feature was first proposed in software engineering, the field of FOSD has been developed forcefully. Researchers from different disciplines have contributed to the current state and success of FOSD. Nevertheless, further steps have to be made in order to realize the full potential of FOSD and to accelerate its adoption in industry and its acceptance in other research fields. We have identified and condensed the following list of key issues, which is ordered following the phases of the FOSD process:

- **Domain analysis**: Feature models are useful for the communication between stakeholders and for the automation of the development process. This twofold usage of feature models imposes a challenge for their further development. On the one hand, feature models should be simple so that stakeholders can understand their meaning. On the other hand, it is useful to enrich feature models with further information in order to guide the generation process, e.g., for domain-specific optimization. The way researchers and practitioners deal with this trade-off will be significant for the success of FOSD. Open issues are: What kind of model (granularity) is needed? Is a single model enough or do we need multiple mapped models? What kind of information is useful and how is this information presented and used?
• **Domain design and specification:** The specification of the structure and behavior of features and their interactions is an important activity that has received too little attention in the past. What are the best techniques for modeling and specifying the desired behavioral and structural properties of features as well as their intended and accidental interactions? How can we verify that a composition of features satisfies the specification of the individual features and of the desired product? Answering these questions will be crucial to convincing practitioners of the usefulness of FOSD.

• **Domain implementation:** Researchers have developed many languages and tools for the implementation of features. Each language and tool has individual strengths and weaknesses, but researchers only begin to understand the implications for successful FOSD, not to speak of the average user. A further problem is that, typically, the different languages and tools do not interoperate well and cannot be integrated with tools from other phases of the FOSD process. Recently, first attempts have been made to unify different implementation approaches and to integrate them into the FOSD process. Researchers should follow this line of work. Furthermore, the idea of factoring feature interaction code via lifters and derivatives is promising, but it is not clear whether this approach scales to large numbers of features with many interactions or whether we need something completely different.

• **Product configuration and generation:** In recent years, techniques and tools from the fields of feature modeling and feature implementation begin to converge toward a unified framework for FOSD. The main challenges are how to specify, verify, and test the correctness of software products generated from product lines, ideally without generating all products. A key problem of FOSD is that the flexibility of feature composition leads to a progressively increasing number of variants, which must be handled by programmers and tools. Further important issues are how to represent domain knowledge, provided by domain experts, and how to generate efficient software products based on this information.

• **FOSD theory:** Finally, first formal models of FOSD are being developed, mostly focusing on specific issues. How are their theories related? How can they be integrated into a unified theory that describes all structures and mechanisms used in the FOSD process? A unified theory of FOSD is the prerequisite for realizing the vision of FOSD: the generation of efficient and correct software on the basis of a set of feature artifacts and a user’s feature selection.

Of course, there are further issues and details that have to be addressed. However, we believe the above list forms an important subset to guide further research.
6 CONCLUSION

The goal of this article has been to provide an overview of FOSD. Beginning from the roots of FOSD, we have systematically summarized several promising works on FOSD. We hope that our survey will help to reveal connections between different approaches to FOSD, to identify and address open issues, and to guide further work and cooperations between different researchers. The ultimate goal is to establish an identity of the FOSD community and to drive further the convergence of work on FOSD.

ACKNOWLEDGMENTS

We thank Don Batory, Krzysztof Czarnecki, Christian Lengauer, Jörg Liebig, Roberto Lopez-Herrejon, Wolfgang Scholz, and Salvador Trujillo for their helpful comments on earlier drafts of this paper. Furthermore, we acknowledge Wolfgang Scholz as inventor of the list example. The first author’s research is sponsored in part by the German Science Foundation (DFG), # AP 206/2-1.

REFERENCES


**ABOUT THE AUTHORS**

**Sven Apel** is a post-doctoral associate at the Chair of Programming at the University of Passau, Germany. He received a Ph.D. in Computer Science from the University of Magdeburg, Germany in 2007. His research interests include advanced programming paradigms, software product lines, and algebra for software construction. He can be reached at apel@uni-passau.de. See also [http://www.infosun.fim.uni-passau.de/cl/staff/apel/](http://www.infosun.fim.uni-passau.de/cl/staff/apel/).

**Christian Kästner** is a Ph.D. student in Computer Science at the University of Magdeburg, Germany. His research interests include languages and tools for software product lines and (virtual) separation of concerns. He can be reached at kaestner@iti.cs.uni-magdeburg.de. See also [http://wwwiti.cs.uni-magdeburg.de/~ckaestne/](http://wwwiti.cs.uni-magdeburg.de/~ckaestne/).