Feature-Oriented Software Development

A Short Tutorial on Feature-Oriented Programming, Virtual Separation of Concerns, and Variability-Aware Analysis*

Christian Kästner[1] and Sven Apel[2]

[1] Philipps University Marburg, Germany
[2] University of Passau, Germany

Abstract. *Feature-oriented software development* is a paradigm for the construction, customization, and synthesis of large-scale and variable software systems, focusing on structure, reuse and variation. In this tutorial, we provide a gentle introduction to software product lines, feature oriented programming, virtual separation of concerns, and variability-aware analysis. We provide an overview, show connections between the different lines of research, and highlight possible future research directions.

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 *variants* of the software that can be *tailored* to the needs of the user and the application scenario. Typically, from a set of features, many different software variants can be generated that share common features and differ in other features. The set of software systems generated from a set of features make up a *software product line* [28, 75].

FOSD aims essentially at three properties: structure, reuse, and variation. Developers use the concept of a feature to structure the design and code of a software system. Features are the primary units of reuse in FOSD. The variants of a software system vary in the features they contain. FOSD shares goals with other software development paradigms, such as stepwise and incremental software development [74, 98], aspect-oriented software development [36], component-based software engineering [88], and alternative flavors of software product line engineering [28, 75], the differences of which are discussed elsewhere [4]. Historically, FOSD has emerged from different lines of research in programming

* These tutorial notes share text with previous publications on feature-oriented software development [3, 4, 47, 49, 93].

```
1  static int __rep_queue_filedone(dbenv, rep, rfp)
2    DB_ENV *dbenv;
3    REP *rep;
4    __rep_fileinfo_args *rfp; {
5  #ifndef HAVE_QUEUE
6    COMPQUIET(rep, NULL);
7    COMPQUIET(rfp, NULL);
8    return (__db_no_queue_am(dbenv));
9  #else
10   db_pgno_t first, last;
11   u_int32_t flags;
12   int empty, ret, t_ret;
13 #ifdef DIAGNOSTIC
14   DB_MSGBUF mb;
15 #endif
16   // over 100 further lines of C code
17 #endif
18 }
```

Fig. 1. Code excerpt of Oracle's Berkeley DB.

languages, software architecture, and modeling; it combines results from feature modeling, feature interaction analysis, and various implementation forms for features [4].

In practice, software product lines are often implemented with build systems and conditional compilation. Hence, developers see code fragments as exemplified in Figure 1, in which code fragments belonging to features are wrapped by #ifdef and #endif directives of the C preprocessor. For a given feature selection, the preprocessor generates tailored code by removing code fragments not needed. Such preprocessor usage is dominant in practice; for example, in HP's product line of printer firmware over 2 000 features are implemented this way, in the Linux kernel over 10 000 features. Although common, such implementations are rather ad-hoc, violate the principle of separation of concerns, and are error prone and difficult to debug; preprocessors are heavily criticized in literature [1, 32, 34, 49, 86, and others]. Especially, if features are scattered and tangled in large-scale programs (or even already at smaller scale as illustrated with the embedded operating system FemtoOS in Fig. 2), such problems quickly become apparent.

FOSD generally seeks more disciplined forms of feature implementation that are easier to maintain and to reason about. Researchers have investigated different strategies for better feature implementations. In this tutorial, we describe two important approaches. First, *feature-oriented programming* follows a language-based composition approach, in which features are implemented in separate implementation units and composed on demand. In contrast, work on *virtual separation of concerns* stays close to the annotation-based approach of preprocessors, but builds upon a disciplined foundation and provides tool support to support reasoning and navigation.

The ability to combine features and derive different variants yields enormous flexibility but also introduces additional problems related to complexity. From n features, we can derive up to 2^n distinct variants (with 33 features, that's more than the number of humans on the planet; with 320 features, that's more

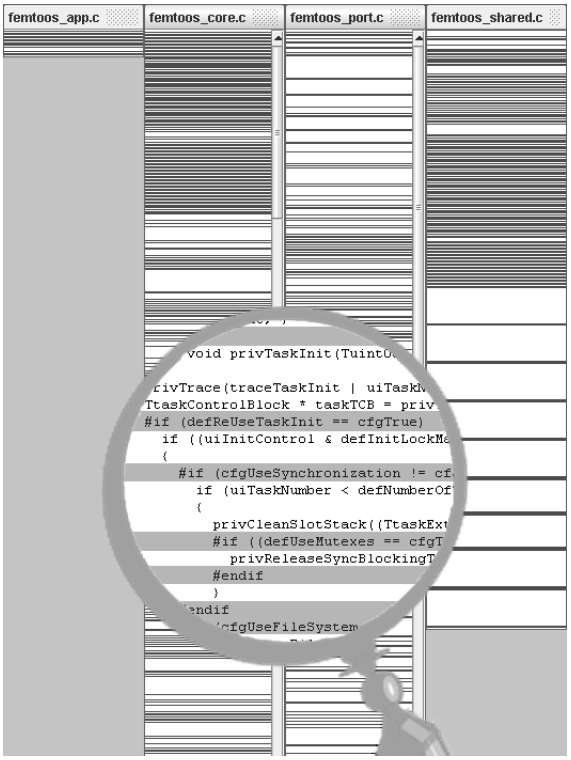

Fig. 2. Preprocessor directives in the code of Femto OS: Black lines represent preprocessor directives such as #ifdef, white lines represent C code, comment lines are not shown [49].

than the estimated number of atoms in the universe). Instead of a single product, product-line developers implement millions of variants in parallel. To support them in dealing with this complexity and to prevent or detect errors (even those that occur only in one variant with a specific feature combination, out of millions), many researchers have proposed means for *variability-aware analysis* that lifts existing analyses to the product-line world. So far, variability-aware analysis has been explored, for example, for type checking, parsing, model checking, and verification. Instead of analyzing each of millions of variants in a brute-force fashion, variability-aware analysis seeks mechanisms to analyze the entire product line. We introduce the idea behind variability-aware analysis and illustrate it with the example of type checking, both for annotations and composition.

This tutorial gives a gentle introduction to FOSD. It is structured as follows: First, we introduce product lines, such as feature models and the process of domain engineering. Second, we exemplify feature-oriented programming with FeatureHouse to separate the implementation of features into distinct modules. Third, we introduce the idea of virtual separation of concerns, an approach that,

instead of replacing preprocessors, disciplines them and provides mechanisms to emulate modularity through dedicated tool support. Finally, we introduce variability-aware analysis by means of the example of type checking and illustrate the general concept behind it.

In contrast to our previous survey on feature-oriented software development [4], which connected different works around the FOSD community, in this tutorial, we take a more practical approach, focus on concepts relevant for implementers, and recommend relevant tools. Additionally, we repeat all relevant background about product-line engineering and feature modeling to make the tutorial more self-contained. Furthermore, we provide a broader picture and a new classification for variability-aware analysis strategies.

2 Software product lines: The basics

Traditionally, software engineering has focused on developing individual software systems, one system at a time. A typical development process starts with analyzing the requirements of a customer. After several development steps – typically some process of specification, design, implementation, testing, and deployment – a single software product is the result. In contrast, *software product line engineering* focuses on the development of *multiple* similar software systems in one domain from a common code base [14, 75]. Although the resulting software products are similar, they are each tailored to the specific needs of different customers or to similar but distinct use cases. We call a software product derived from a software product line a *variant*.

Bass et al. define a software product line as *"a set of software-intensive systems sharing a common, managed set of features that satisfy the specific needs of a particular market segment or mission and that are developed from a common set of core assets in a prescribed way"* [14]. The idea of developing a set of related software products in a coordinated fashion (instead of each starting from scratch or copying and editing from a previous product) can be traced back to concepts of *program families* [42, 74].

Software product lines promise several benefits compared to individual development [14, 75]: Due to co-development and systematic reuse, software products can be produced faster, with lower costs, and higher quality. A decreased time to market allows companies to adapt to changing markets and to move into new markets quickly. Especially in embedded systems, in which resources are scarce and hardware is heterogeneous, efficient variants can be tailored to a specific device or use case [19, 75, 80, 91]. There are many companies that report significant benefits from software product lines. For example, Bass et al. summarize that, with software product lines, Nokia can produce 30 instead of previously 4 phone models per year; Cummins, Inc. reduced development time for a software for a new diesel engine from one year to one week; Motorola observed a 400 % increase in productivity; and so forth [14].

2.1 Domain engineering and application engineering

The process of developing an entire software product line instead of a single application is called *domain engineering*. A software product line must fulfil not only the requirements of a single customer but the requirements of multiple customers in a domain, including both current customers and potential future customers. Hence, in domain engineering, developers analyze the entire application domain and its potential requirements. From this analysis, they determine commonalities and differences between potential variants, which are described in terms of features. Finally, developers design and implement the software product line such that different variants can be constructed from common and variable parts.

In this context, a feature is a first-class domain abstraction, typically an end-user visible increment in functionality. In addition to features that add functionality, it is also common to have alternative features for the same functionality with different nonfunctional properties (e.g., a fast versus an energy-saving sorting algorithm). We discuss different notions of the term "feature" elsewhere [4].

Czarnecki and Eisenecker distinguish between *problem space* and *solution space* [30]. The problem space comprises domain-specific abstractions that describe the requirements on a software system and its intended behavior. Domain analysis, as a part of domain engineering, takes place in the problem space, and its results are documented in terms of features. The solution space comprises implementation-oriented abstractions, such as code artifacts. Between features in the problem space and artifacts in the solution space, there is a mapping that describes which artifact belongs to which feature. Depending on the implementation approach and the degree of automation, this mapping can have different forms and complexities, from simple implicit mappings based on naming conventions to complex machine-processable rules encoded in generators, including preprocessors and composition tools [30].

Application engineering is the process of deriving a single variant tailored to the requirements of a specific customer from a software product line, based on the results of domain engineering. Ideally, the customer's requirements can be mapped to features identified during domain engineering (problem space), so that the variant can be constructed from existing common and variable parts of the product line's implementation (solution space). FOSD strives for a form of product-line development in which all implementation effort is part of domain engineering so that application engineering can be reduced to requirements analysis and automated code generation.

Typically, a software product line targets a specific domain, such as operating systems for mobile phones, control software for diesel engines, and embedded databases. The *scope* of a software product line describes which variability is offered and which kind of variants the product line can produce. A software product line with a narrow scope is easier to develop, but less flexible (it provides only few, very similar variants). The wider the scope is, the higher is the development effort, but the more flexibility a software product line can offer. Selecting the right scope of a product line is a difficult design, business, and strategy decision. In

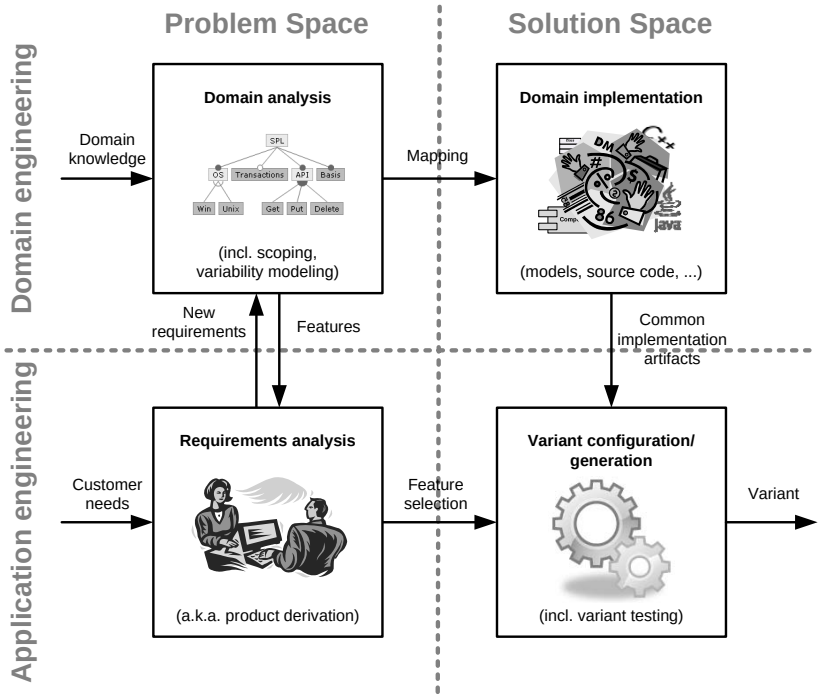

Fig. 3. An (idealized) overview of domain engineering and application engineering (adapted from [30] to FOSD).

practice, the scope is often iteratively refined; domain engineering and application engineering are rarely strictly sequential and separated steps. For example, it is common not to implement all features upfront, but incrementally, when needed. Furthermore, requirements identified in domain engineering may be incomplete, so new requirements arise in application engineering, which developers must either feed back into the domain-engineering process or address with custom development during the application engineering of a specific variant [30].

Domain engineering and application engineering describe a general process framework as summarized in Figure 3. For each step, different approaches, formalisms, and tools can be used. For example, there are different product-line–scoping approaches (see a recent survey [45]), different domain analysis methods [30,40,46,75, and many others], different mechanisms to model variability (see Sec. 2.2), different implementation mechanisms (our focus in Sec. 3 and 4), and different approaches to derive a variant based on customer requirements [78, 82, and others].

2.2 Variability modeling

During domain analysis, developers determine the scope of the software product line and identify its common and variable features, which they then document in a variability model. We introduce variability models, because they are central not only for documenting variability in the problem space, but also for many implementation approaches, for automated reasoning and error detection, and for automated generation of variants. There are several different variability-modeling approaches (see Chen et al. [26] for an overview). We focus on FODA-style *feature models* [30, 46], because they are well known and broadly used in research and practice; other variability models can be used similarly.

A feature model describes a set of features in a domain and their relationships. It describes which features a product line provides (i.e., its scope), which features are optional, and in which combination features can be selected in order to derive variants. With a selection of features (a subset F of all features), we can specify a variant (e.g., "the database variant for Linux, with transactions, but without a B-Tree"). Not all feature combinations may make sense, for example, two features representing different operating systems might be mutually exclusive. A feature model describes such dependencies. A feature selection that fulfils all constraints is *valid* ("F is valid").

In practice, feature models contain hundreds or thousands of features.[1] The number of potential variants can grow exponentially with the number of features. In theory, a software product line with n independent optional features can produce 2^n variants. In practice, many dependencies between features reduce the number of valid feature selections, but nevertheless, most software product lines give rise to millions or billions of valid feature selections.

A typical graphical representation of features and their dependencies is a *feature diagram* [46], as exemplified in Figure 4. A feature diagram represents features in a hierarchy. Different edges between features describe their relationships: A filled bullet describes that a feature is mandatory and must be selected whenever its parent feature is selected. In contrast, a feature connected with an empty bullet is optional. Multiple child features connected with an empty arc are alternative (mutually exclusive); exactly one child feature needs to be selected when the parent feature is selected. From multiple child features connected with a filled arc, at least one must be selected, but it is also possible to select more than one. Dependencies that cannot (or should not) be expressed with the hierarchical structure may be provided as additional cross-tree constraints in the form of a propositional formula. In Figure 4, we show nine features from the core of a fictional database product line. Each variant must contain the features Database, Base, OS, and Storage, but feature Transactions is optional, so variants may or may not include it; each variant must have exactly one operating-system feature, either Windows or Linux; each variant must contain at least one storage structure; finally, a cross-tree constraint specifies

[1] For example, Bosch's product line of engine-control software has over 1 000 features [87], HP's Owen product line has about 2 000 features [79], and the Linux kernel has over 10 000 features [90].

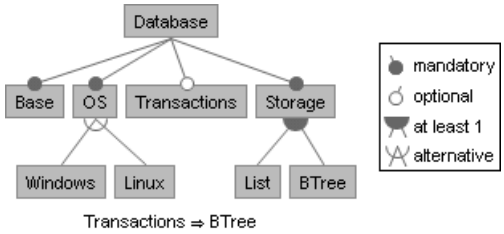

Fig. 4. Feature-diagram example of a small database product line.

that TRANSACTIONS are supported only if also feature B-TREE is selected. In this example, ten feature selections are valid.

Alternative to the graphical notation, dependencies between features can be expressed entirely by a propositional formula. Each feature corresponds to a Boolean variable that is *true* when selected and *false* otherwise. The propositional formula evaluates to *true* for all valid feature selections. Feature diagrams can be transformed into propositional formulas with some simple rules [15]. For example, the feature diagram from Figure 4 is equivalent to the following propositional formula:

$$\begin{aligned}&\text{DATABASE} \wedge (\text{BASE} \Leftrightarrow \text{DATABASE}) \wedge (\text{OS} \Leftrightarrow \text{DATABASE}) \wedge \\ &(\text{TRANSACTIONS} \Rightarrow \text{DATABASE}) \wedge (\text{STORAGE} \Leftrightarrow \text{DATABASE}) \wedge \\ &(\text{WINDOWS} \vee \text{LINUX} \Leftrightarrow \text{OS}) \wedge \neg(\text{WINDOWS} \wedge \text{LINUX}) \wedge \\ &(\text{LIST} \vee \text{B-TREE} \Leftrightarrow \text{STORAGE}) \wedge (\text{TRANSACTIONS} \Rightarrow \text{B-TREE})\end{aligned}$$

Representing feature models as propositional formulas has the advantage that we can reason about them automatically, which is essential for variability-aware analysis, as we discuss in Section 5. With simple algorithms or with automated reasoning techniques – including Boolean-satisfiability-problem solvers (SAT solvers), constraint-satisfaction-problem solvers, and binary decision diagrams – we can efficiently answer a series of questions, including "Has this feature model at least one valid selection (i.e., is the formula satisfiable)?" and "Is there a valid feature selection that includes feature X but not feature Y?" Even though some of these algorithms are NP-complete, SAT solvers and other reasoners can answer queries efficiently for practical problems, even for very large feature models [67, 68, 94]. For further details, see a recent survey on automated analysis operations and tools [18].

Tooling. There are many languages and tools to manage feature models or draw feature diagrams, ranging from dozens of academic prototypes to fully fledged commercial systems such as *Gears*[2] and *pure::variants*.[3] For a research setting,

[2] `http://www.biglever.com/solution/product.html`

[3] `http://www.pure-systems.com`; a limited community edition is available free of charge, and the authors are open for research collaborations.

we recommend FeatureIDE, an Eclipse plugin that (among others) provides a sophisticated graphical feature-model editor and supporting tools [57]. Our graphics of feature diagrams (Fig. 3 and 4) have been exported from FeatureIDE. FeatureIDE includes many facilities for reasoning about features using a SAT solver, following the described translation to propositional formulas. FeatureIDE is open source, and also isolated parts such as the reasoning engine can be reused; contributions are encouraged. FeatureIDE is available at `http://fosd.net/fide`.

2.3 What is feature-oriented software development?

The concept of a feature is useful to describe commonalities and variabilities in the analysis, design, and implementation of software systems. FOSD is a paradigm that encourages 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, to structure software along these variabilities and commonalities, and to guide the testing process. 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. Our main goal 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.

3 Feature-oriented programming

The key idea of feature-oriented programming is to decompose a system's design and code along the features it provides [16, 77]. Feature-oriented programming follows a disciplined language-oriented approach, based on feature composition.

3.1 Collaboration-based design

A popular technique for decomposing feature-oriented systems is *collaboration-based design* [85]. In Figure 5, we show a sample collaboration-based design of a simple object-oriented expression evaluator. A *collaboration* is a set of program elements that cooperate systematically to implement a feature. In an object-oriented world, a collaboration comprises typically multiple classes and even only fragments of classes. The top-most collaboration (Expr) consists of three classes: Expr an abstract class for representing expressions, Val for representing

literals, and Add for representing addition. Each class defines a single operation toString for pretty printing. The collaboration Eval adds the new operation eval, which evaluates an expression. Evaluation is a crosscutting concern because eval must be defined by adding a method to each of the three classes. A collaboration bundles these changes.

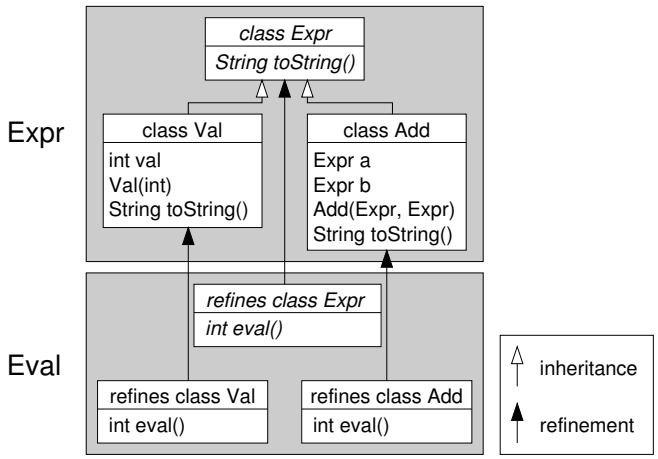

Fig. 5. Collaboration-based design of a simple expression evaluator.

3.2 Feature modules

In feature-oriented programming, each collaboration implements a feature and is called a *feature module* [10, 16]. Different combinations of feature modules satisfy different needs of customers or application scenarios. Figure 5 illustrates how features crosscut the given hierarchical (object-oriented) program structure. In contemporary feature-oriented–programming languages and tools, such as AHEAD [16], FeatureC++ [9], FeatureHouse [7], or Fuji [8], collaborations are represented by file-system directories, called *containment hierarchies*, and classes and their refinements are stored in files. Features are selected by name via command-line parameters or graphical tools. In Figure 6, we show a snapshot of the containment hierarchies and the feature model of the simple expression evaluator in FeatureIDE.

A feature module refines the content of a base program either by adding new elements or by modifying and extending existing elements. The order in which features are applied is important; earlier features in the sequence may add elements that are refined by later features.

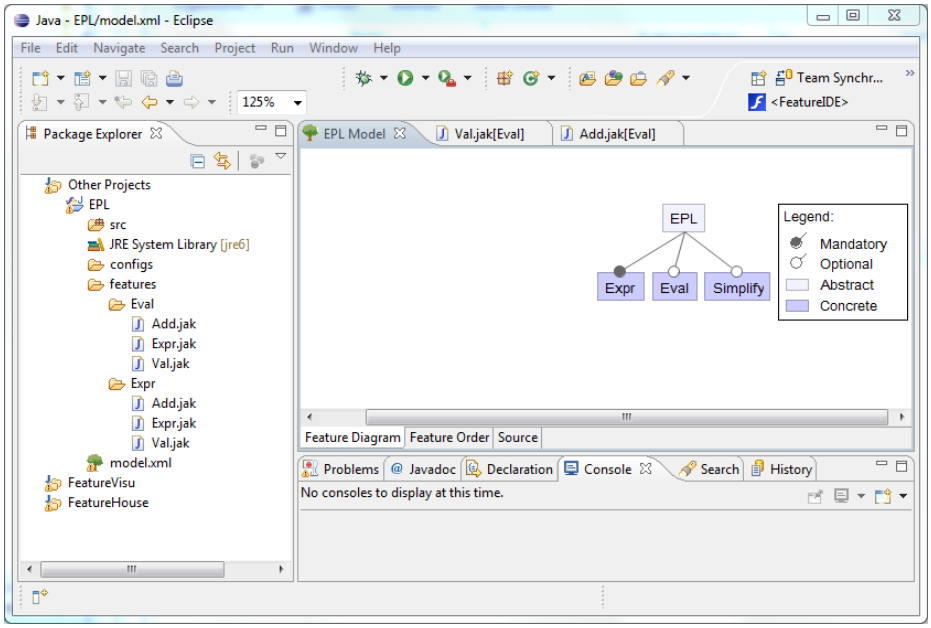

Fig. 6. Containment hierarchy (left) and feature model (right) of the expression-evaluator example.

3.3 Jak

Jak is an extension of Java for feature-oriented programming [16]. Figure 7 depicts the Jak implementation of an extended version of the collaboration-based design of Figure 5.

Feature Expr represents the base program. It defines class Expr, along with two terms: Val for integer literals and Add for addition. It also defines a single operation toString for pretty printing.

Feature Eval adds the new operation eval, which evaluates an expression. The feature module contains three class refinements (partial classes, using the keyword refines) that extend other classes by introducing additional methods. During composition a class is composed with all its refinements.

Feature Mult introduces the new class Mult and refines a previously defined method in class Add to fix operator precedence. Refining a method is similar to method overriding; the new version of the method may call the old version using Jak's keyword Super.

Finally, features Eval and Mult are each designed to extend Expr. However, they are not completely orthogonal. The combination of a new variant and a new operation, creates a "missing piece" that must be filled in to create a complete program. We thus define an additional feature, called *lifter* [77] or *derivative* [65], that defines how each feature should be extended in the presence of the others.

Feature Expr

```
1  abstract class Expr {
2    abstract String toString();
3  }
4  class Val extends Expr {
5    int val;
6    Val(int n) { val = n; }
7    String toString() { return String.valueOf(val); }
8  }
9  class Add extends Expr {
10   Expr a; Expr b;
11   Add(Expr e1, Expr e2) { a = e1; b = e2; }
12   String toString() { return a.toString() + "+" + b.toString(); }
13 }
```

Feature Eval refines Expr

```
14 refines class Expr {
15   abstract int eval();
16 }
17 refines class Val {
18   int eval() { return val; }
19 }
20 refines class Add {
21   int eval() { return a.eval() + b.eval(); }
22 }
```

Feature Mult refines Expr

```
23 class Mult extends Expr {
24   Expr a; Expr b;
25   Mult(Expr e1, Expr e2) { a = e1; b = e2; }
26   String toString() { return "(" + a.toString() + "*" + b.toString() + ")";
27   }
28 }
29 refines class Add {
30   String toString() { return "(" + Super().toString() + ")"; }
31 }
```

Derivative Mult#Eval

```
32 refines class Mult {
33   int eval() { return a.eval() * b.eval(); }
34 }
```

Fig. 7. A solution to the "expression problem" in Jak.

The derivative 'Mult#Eval' is present when both features Mult and Eval are present.

3.4 AHEAD

AHEAD is an architectural model of feature-oriented programming [16]. With AHEAD, each feature is represented by a containment hierarchy, which is a directory that maintains a substructure organizing the feature's artifacts (cf. Fig. 6). Composing features means composing containment hierarchies and, to this end, composing corresponding artifacts recursively by name and type (see Fig. 10 for an example), much like the mechanisms of hierarchy combination [70, 89], mixin composition [20, 24, 37, 38, 85], and superimposition [21, 22]. In contrast to these earlier approaches, for each artifact type, a different implementation of the

composition operator '•' has to be provided in AHEAD (i.e., different tools that perform the composition, much like Jak for Java artifacts). The background is that a complete software system does not just involve Java code. It also involves many non-code artifacts. For example, the simple expression evaluator of Figure 7 may be paired with a grammar specification, providing concrete syntax for expressions, and documentation in XHTML. For grammar specifications and XML based languages, the AHEAD tool suite has dedicated composition tools.

Bali. *Bali* is a tool for synthesizing program-manipulation tools on the basis of extensible grammar specifications [16]. It allows a programmer to define a grammar and to refine it subsequently, in a similar fashion to class refinements in Jak. Figure 8 shows a grammar and a grammar refinement that correspond to the Jak program above. The base program defines the syntax of arithmetic expressions that involve addition only. We then refine the grammar by adding support for multiplication.

Feature Expr

```
1 Expr: Val | Expr Oper Expr;
2 Oper: '+';
3 Val: INTEGER;
```

Feature Mult refines Expr

```
4 Oper: Super.Oper | '*';
```

Fig. 8. A Bali grammar with separate features for addition and multiplication.

Bali is similar to Jak in its use of keyword Super: Expression Super.Oper refers to the original definition of Oper.

Xak. *Xak* is a language and tool for composing various kinds of XML documents [2]. It enhances XML by a module structure useful for refinement. This way, a broad spectrum of software artifacts can be refined à la Jak, (e.g., UML diagrams, build scripts, service interfaces, server pages, or XHTML).

Figure 9 depicts an XHTML document that contains documentation for our expression evaluator. The base documentation file describes addition only, but we refine it to add a description of evaluation and multiplication as well. The tag xak:module labels a particular XML element with a name that allows the element to be refined by subsequent features. The tag xak:extends overrides an element that has been named previously, and the tag xak:super refers to the original definition of the named element, just like the keyword Super in Jak and Bali.

AHEAD tool suite. Jak, Xak, and Bali are each designed to work with a particular kind of software artifact. The AHEAD tool suite brings these separate

Feature Expr

```
<html xmlns:xak="http://www.onekin.org/xak" xak:artifact="Expr" xak:type="xhtml">
  <head><title>A Simple Expression Evaluator</title></head>
  <body bgcolor="white">
    <h1 xak:module="Contents">A Simple Expression Evaluator</h1>
    <h2>Supported Operations</h2>
    <ul xak:module="Operations">
      <li>Addition of integers</li>
      <!-- a description of how integers are added -->
    </ul>
  </body>
</html>
```

Feature Eval refines Expr

```
<xak:refines xmlns:xak="http://www.onekin.org/xak" xak:artifact="Eval" xak:type="xhtml">
  <xak:extends xak:module="Contents">
    <xak:super xak:module="Contents"/>
    <h2>Evaluation of Arithmetic Expressions</h2>
    <!-- a description of how expressions are evaluated -->
  </xak:extends>
</xak:refines>
```

Feature Mult refines Expr

```
<xak:refines xmlns:xak="http://www.onekin.org/xak" xak:artifact="Mult" xak:type="xhtml">
  <xak:extends xak:module="Operations">
    <xak:super xak:module="Operations"/>
    <li>Multiplication of integers</li>
    <!-- a description of how integers are multiplied -->
  </xak:extends>
</xak:refines>
```

Fig. 9. A Xak/XHTML document with separate features for addition, evaluation, and multiplication.

tools together into a system that can handle many different kinds of software artifacts.

In AHEAD, a piece of software is represented as a directory of files. Composing two directories together will merge subdirectories and files with the same name. AHEAD will select different composition tools for different kinds of files. Merging Java files will invoke Jak to refine the classes, whereas merging XML files will invoke Xak to combine the XML documents, and so on, as illustrated in Figure 10.

3.5 FeatureHouse

Recently, following the philosophy of AHEAD, the FeatureHouse tool suite has been developed that allows programmers to enhance given languages rapidly with support for feature-oriented programming (e.g., C#, C, JavaCC, Haskell, Alloy, and UML [7]).

FeatureHouse is a framework for software composition supported by a corresponding tool chain. It provides facilities for feature composition based on a language-independent model of software artifacts and an automatic plugin mechanism for the integration of new artifact languages. FeatureHouse improves

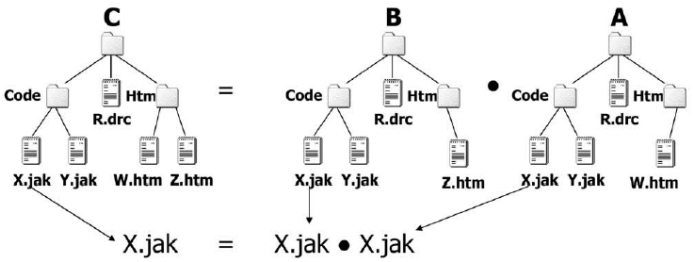

Fig. 10. Composing containment hierarchies by superimposition [16].

over prior work on AHEAD in that it implements language-independent software composition.

Feature structure trees. FeatureHouse relies on a general model of the structure of software artifacts, called the *feature structure tree* (FST) model. An FST represents the essential structure of a software artifact and abstracts from language-specific details. For example, an artifact written in Java contains packages, classes, methods, and so forth, which are represented by nodes in its FST; a Haskell program contains equations, algebraic data types, type classes, etc., which contain further elements; a makefile or build script consists of definitions and rules that may be nested.

Each node of an FST has (1) a name that is the name of the corresponding structural element and (2) a type that represents the syntactic category of the corresponding structural element. For example, a Java class Foo is represented by a node Foo of type Java class. Essentially, an FST is a stripped-down abstract syntax tree (AST): it contains only information that is necessary for the specification of the modular structure of an artifact and for its composition with other artifacts. The inner nodes of an FST denote modules (e.g., classes and packages) and the leaves carry the modules' content (e.g., method bodies and field initializers). We call the inner nodes *nonterminals* and the leaves *terminals*. For illustration, in Figure 11, we depict on the left side the FST of concerning class Val of feature Expr.

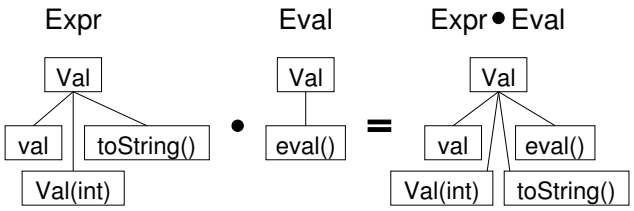

Fig. 11. Superimposition of feature structure trees (excerpt of the expression example).

What code elements are represented as inner nodes and leaves? This depends on the language and on the level of *granularity* at which software artifacts are to be composed [50]. Different granularities are possible and might be desired in different contexts. For Java, we could represent only packages and classes but not methods or fields as FST nodes (a coarse granularity), or we could also represent statements or expressions as FST nodes (a fine granularity). In any case, the structural elements not represented in the FST are text content of terminal nodes (e.g., the body of a method). In our experience, the granularity of Figure 11 is usually sufficient for composition of Java artifacts.

Superimposition. The composition of software artifacts proceeds by the superimposition of the corresponding FSTs, denoted by '•'. Much like in AHEAD, two FSTs are superimposed by merging their nodes, identified by their names, types, and relative positions, starting from the root and descending recursively. Figure 11 illustrates the process of FST superimposition with the expression example (only concerning class Val).

Generally, the composition of two leaves of an FST that contain further content demands a special treatment. The reason is that the content is not represented as a subtree but as plain text. Method bodies are composed differently from fields, Haskell functions, or Bali grammar productions. The solution is that, depending on the artifact language and node type, different rules for the composition of terminals are used. Often simple rules such as replacement, concatenation, specialization, or overriding suffice, but the approach is open to more sophisticated rules known from multi-dimensional separation of concerns [71] or software merging [69]. For example, we merge two method bodies via overriding, in which Super defines how the bodies are merged, much like in Jak.

Generation and Automation. New languages can be plugged easily into FeatureHouse. The idea is that, although artifact languages are very different, the process of software composition by superimposition is very similar. For example, the developers of AHEAD/Jak [16] and FeatureC++ [9] have extended the artifact languages Java and C++ by constructs (e.g., refines or Super) and mechanisms for composition. They have each implemented a parser, a superimposition algorithm, and a pretty printer[4] – all specific to the artifact language. We have introduced the FST model to be able to express superimposition independently of an artifact language [11].

In FeatureHouse, we automate the integration of further languages and base it largely on the languages' grammars. This allows us to *generate* most of the code that must otherwise be provided and integrated manually (parser, adapter, pretty printer) and to experiment with different representations of software artifacts. Our tool FSTGenerator expects the grammar of the language to be integrated in a specific format, called FeatureBNF, and generates a parser,

[4] With 'pretty printer' we refer to a tool, also known as unparser, that takes a parse tree or an FST and generates source code.

adapter, and pretty printer accordingly. Using a grammar written in FeatureBNF, FSTGenerator generates (a) an LL(k) parser that directly produces FST nodes and (b) a corresponding pretty printer. After the generation step, composition proceeds as follows: (1) the generated parser receives artifacts written in the target language and produces one FST per artifact; (2) FeatureHouse performs the composition; and (3) the generated pretty printer writes the composed artifacts to disk. For the composition of the content of terminal nodes, we have developed and integrated a library of composition rules (e.g., rules for method overriding or for the concatenation of the statements of two constructors). Figure 12 illustrates the interplay between FSTGenerator and FeatureHouse;

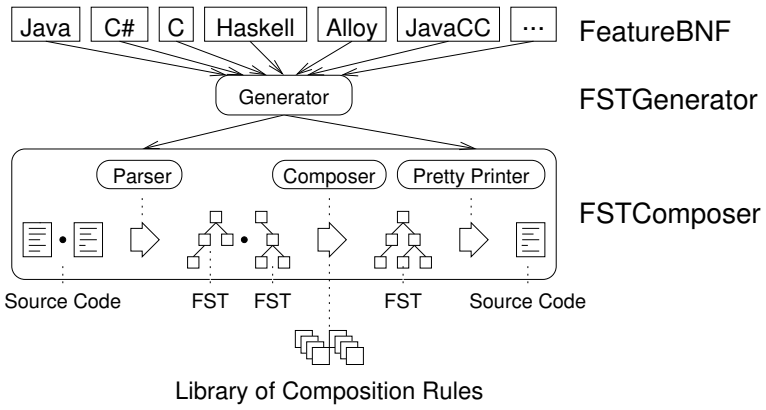

Fig. 12. The architecture of FeatureHouse

A detailed description of FSTGenerator and FeatureBNF is available elsewhere [7].

Tooling. Both AHEAD[5] and FeatureHouse[6] are available for experimentation including several examples. Both are command-line tools. FeatureIDE provides a graphical front end in Eclipse, with corresponding editors for Jak, a mapping from features to feature modules, automatic composition of selected features in the background, generation of collaboration diagrams, and much more [57, 62]. FeatureIDE ships with AHEAD and FeatureHouse and several example projects, ready to explore. After a developer graphically configures the desired features, FeatureIDE automatically calls the corresponding composition tools. It is likely the easiest way to try AHEAD or FeatureHouse, for developers familiar with Eclipse. Recently, Batory contributed even a video tutorial on FeatureIDE.[7]

[5] `http://www.cs.utexas.edu/users/schwartz/ATS.html`
[6] `http://fosd.net/fh`
[7] `http://www.cs.utexas.edu/users/dsb/cs392f/Videos/FeatureIDE/`

4 Virtual separation of concerns

Recently, several researchers have taken a different path to tackle more disciplined product-line implementations. Instead of inventing new languages and tools that support feature decomposition, they stay close to the concept of conditional compilation with preprocessors, but improve it at a tooling level. The goal is to keep the familiar and simple mechanisms of annotating code fragments in a common implementation (e.g., as with the C preprocessor), but to emulate modularity with tool support and to provide navigation facilities as well as error diagnostics. We work around the limitations for which traditional preprocessors are typically criticized.

4.1 Variability implementation with preprocessors

Conditional-compilation mechanisms of preprocessors provide an easy strategy to implement compile-time variability in product lines. The concept is simple: Developers annotate code fragments with feature expressions. Subsequently the preprocessor removes certain annotated code fragments before compilation, depending on the feature selection.

To introduce preprocessors, we exemplify a preprocessor-based implementation of the "expression problem" from Figure 7 in Figure 13. We use the preprocessor Antenna,[8] which was developed for Java code on mobile platforms. Conditional compilation in Antenna uses almost the same notation as in the C preprocessor, but preprocessor directives are written in comments, to not interfere with existing tool support for Java code. Variable code fragments are framed with #ifdef and #endif directives. In a feature-oriented context, the #ifdef directives refer to features from the feature model. If the corresponding feature is not selected, the code fragment between the #ifdef and the #endif directive is removed before compilation. Furthermore, #ifdef directives may be nested, so that code is only included if multiple features are selected; for example, Line 45 in Figure 13 is only included if features MULT and EVAL are both selected (equivalent to derivative modules discussed in Section 3.3)

4.2 Disciplined preprocessor usage

A main problem of traditional (lexical) preprocessors, such as the C preprocessor, is that they are oblivious to the underlying host language and the variability specification. It is possible to annotate individual tokens such as a closing bracket, leading to hard-to-find syntax errors. For the same reason, parsing unpreprocessed code for analysis is a difficult task (a parser can hardly foresee all possibilities how the preprocessor is used) [17, 39, 56, 64, 73]. The mapping between features in a feature model and `#ifdef` flags is not checked, hence a typo in a flag name leads to never compiling this code fragment [90]. In feature-oriented programming, these problems do not occur, because the underlying language allows only disciplined

[8] `http://antenna.sourceforge.net/`

```
1  abstract class Expr {
2    abstract String toString();
3  //#ifdef EVAL
4    abstract int eval();
5  //#endif
6  }
7
8  class Val extends Expr {
9    int val;
10   Val(int n) { val = n; }
11   String toString() { return String.valueOf(val); }
12 //#ifdef EVAL
13   int eval() { return val; }
14 //#endif
15 }
16
17 class Add extends Expr {
18   Expr a; Expr b;
19   Add(Expr e1, Expr e2) { a = e1; b = e2; }
20   String toString() {
21     StringBuffer r=new StringBuffer();
22 //#ifdef MULT
23     r.append("(");
24 //#endif
25     r.append(a.toString());
26     r.append("+");
27     r.append(b.toString());
28 //#ifdef MULT
29     r.append(")");
30 //#endif
31     return r.toString();
32   }
33 //#ifdef EVAL
34   int eval() { return a.eval() + b.eval(); }
35 //#endif
36 }
37
38 //#ifdef MULT
39 class Mult extends Expr {
40   Expr a; Expr b;
41   Mult(Expr e1, Expr e2) { a = e1; b = e2; }
42   String toString() { return "(" + a.toString() + "*" + b.toString() + ")";
43   }
44 //#ifdef EVAL
45   int eval() { return a.eval() * b.eval(); }
46 //#endif
47 }
48 //#endif
```

Fig. 13. A preprocessor-based implementation of the "expression problem" from Figure 7.

usage, but preprocessors are a different story. Overall, the flexibility of lexical preprocessors allows undisciplined use that is hard to understand, to debug, and to analyze.

To overcome the above problems, we require a disciplined use of preprocessors. With disciplined use, we mean that annotations (in the simplest form #ifdef flags) must correspond to feature names in a feature model and that annotations align with the syntactic structure of the underlying language [50, 54, 64]. For example, annotating an entire statement or an entire function is considered disciplined; the annotation aligns with the language constructs of the host language. In contrast, we consider annotating an individual bracket or just the return type of a function as undisciplined. In Figure 14, we illustrate several examples of disciplined and undisciplined annotations from the code of the text editor *vim*. A restriction to disciplined annotations enables easy parsing of the source code [17, 64, 66] and hence makes the code available to automated analysis (including variability-aware analysis, as discussed in Sec. 5). Code with disciplined annotations can be represented in the choice calculus [33], which opens the door for formal reasoning and for developing a mathematical theory of annotation-based FOSD. As a side effect, it guarantees that all variants are syntactically correct [54].

```
1 void tcl_end() {
2 #ifdef DYNAMIC_TCL
3   if (hTclLib) {
4     FreeLibrary(hTclLib);
5     hTclLib = NULL;
6   }
7 #endif
8 }
```

disciplined annotation

```
1   int n = NUM2INT(num);
2 #ifndef FEAT_WINDOWS
3   w = curwin;
4 #else
5   for (w = firstwin; w != NULL; w =
        w->w_next, --n)
6 #endif
7     if (n == 0)
8       return window_new(w);
```

undisciplined annotation (for wrapper)

```
1   if (char2cells(c) == 1
2 #if defined(FEAT_CRYPT) || defined(FEAT_EVAL)
3       && cmdline == 0
4 #endif
5   )
```

undisciplined annotation at expression level

```
1   if (!ruby_initialized) {
2 #ifdef DYNAMIC_RUBY
3     if (ruby_enabled(TRUE)) {
4 #endif
5       ruby_init();
```

undisciplined annotation (if wrapper)

Fig. 14. Examples of disciplined and undisciplined annotations in *vim* [64].

There are different ways to enforce annotation discipline. For example, we can introduce conditional compilation facilities into a programming language, instead of using an external preprocessor, as done in D[9] and rbFeatures [41]. Similarly, syntactic preprocessors allow only transformations based on the underlying structure [23, 66, 97]. Alternatively, we can check discipline after the fact by

[9] http://www.digitalmars.com/d/

running additional analysis tools (however, even though Linux has a script to check preprocessor flags against a feature model, Tartler et al. report several problems in Linux with incorrect config flags as the tool is apparently not used [90]). Finally, in our tool CIDE, we map features to code fragments entirely at the tool level, such that the tool allows only disciplined annotations; hence, a developer is not able to make an undisciplined annotation in the first place [50].

Enforcing annotation discipline limits the expressive power of annotations and may require somewhat higher effort from developers who need to rewrite some code fragments. Nevertheless, experience has shown that the restriction to disciplined annotations are not a serious limitation in practice [17, 54, 64, 96]. Developers can usually rewrite undisciplined annotations locally into disciplined ones – there is even initial research to automate this process [39,56]. Furthermore, developers usually prefer disciplined annotations anyway (and sometimes, e.g., in Linux, have corresponding guidelines), because they understand the threats to code comprehension from undisciplined usage. Liebig et al. have shown that 84 % of all #ifdef directives in 40 substantial C programs are already in a disciplined form [64]. So, we argue that enforcing discipline, at least for new projects, should be a viable path that eliminates many problems of traditional preprocessors.

Disciplined usage of annotations opens annotation-based implementations to many forms of analysis and tool support, some of which we describe in the following. Many of them would not have been possible with traditional lexical preprocessors.

4.3 Views

One of the key motivations of modularizing features (for example, with feature-oriented programming) is that developers can find all code of a feature in one spot and reason about it without being distracted by other concerns. Clearly, a scattered, preprocessor-based implementation, as in Figure 2, does not support this kind of lookup and reasoning, but the core question "what code belongs to this feature" can still be answered by tool support in the form of *views* [44, 58, 84].

With relatively simple tool support, it is possible to create an (editable) view on the source code by hiding all irrelevant code of other features. In the simplest case, we hide files from the file browser in an IDE. Developers will only see files that contain code of certain features selected interactively by the user. This way, developers can quickly explore all code of a feature without global code search.

In addition, views can filter code *within* a file (technically, this can be implemented like code folding in modern IDEs).[10] In Figure 15, we show an example

[10] Although editable views are harder to implement than read-only views, they are more useful since users do not have to go back to the original code to modify it. Implementations of editable views have been discussed intensively in work on database or model-roundtrip engineering. Furthermore, a simple but effective solution, which we apply in our tools, is to leave a marker indicating hidden code [50]. Thus, modifications occur before or after the marker and can be unambiguously propagated to the original location.

```
1  class Stack implements IStack {
2    void push(Object o) {
3  //#ifdef TXN
4      Lock l = lock(o);
5  //#endif
6  //#ifdef UNDO
7      last = elementData[size];
8  //#endif
9      elementData[size++] = o;
10 //#ifdef TXN
11     l.unlock();
12 //#endif
13     fireStackChanged();
14   }
15 //#ifdef TXN
16   Lock lock(Object o) {
17     return LockMgr.lockObject(o);
18   }
19 //#endif
20   ...
21 }
```

(a) original (all features selected)

```
1  class Stack [] {
2    void push([]) {
3      Lock l = lock(o);
4      []
5      l.unlock();
6      []
7    }
8    Lock lock(Object o) {
9      return LockMgr.lockObject(o);
10   }
11   []
12 }
```

(b) view on TXN (hidden code is indicated by '[]', necessary context information is shown in gray italics)

Fig. 15. View emulates separation of concerns [47].

of a code fragment and a view on its feature TRANSACTION (TXN). Note, we cannot simply remove everything that is not annotated by #ifdef directives, because we could end up with completely unrelated statements. Instead, we need to provide some context (e.g., in which class and method is this statement located); in Figure 15, we highlight the context information in gray and italic font. Interestingly, similar context information is also present in modularized implementations in the form of class refinements, method signatures, pointcuts, or extension points.

Beyond views on one or more individual features, (editable) views *on variants* are possible [13, 43, 58]. That is, a tool can show the source code that would be generated for a given feature selection and hide all remaining code of unselected features. With such a view, a developer can explore the behavior of a variant when multiple features interact, without being distracted by code of unrelated features. This goes beyond the power of physical separation with tools such as FeatureHouse, with which the developer has to reconstruct the behavior of multiple components/plug-ins/aspects in her mind. Especially, when many fine-grained features interact, from our experience, views can be a tremendous help. Nevertheless, some desirable properties such as separate compilation or modular type checking cannot be achieved with views.

Hence, views can emulate some advantages of separating features as in feature-oriented programming. Developers can quickly explore all code of a feature and can deliberately navigate between features by switching between different views. We have implemented the described views in our tool CIDE [50]. Instead of a physical separation of features into separate files or directories, views provide a virtual separation, hence the name *virtual separation of concerns*.

```
1  class Stack {
2    void push(Object o
3  //#ifdef TXN
4    , Transaction txn
5  //#endif
6    ) {
7      if (o==null
8  //#ifdef TXN
9        || txn==null
10 //#endif
11       ) return;
12 //#ifdef TXN
13     Lock l=txn.lock(o);
14 //#endif
15     elementData[size++] = o;
16 //#ifdef TXN
17     l.unlock();
18 //#endif
19     fireStackChanged();
20   }
21 }
```

Fig. 16. Java code obfuscated by fine-grained annotations with *cpp*.

```
1 class Stack {
2   void push(Object o, Transaction txn) {
3     if (o==null || txn==null) return;
4     Lock l=txn.lock(o);
5     elementData[size++] = o;
6     l.unlock();
7     fireStackChanged();
8   }
9 }
```

Features: TRANSACTION

Fig. 17. Annotated code represented by background color instead of textual annotation [49].

4.4 Coping with obfuscated source code

Traditional preprocessors have a reputation for obfuscating source code such that the resulting code is difficult to read and maintain. The reason is that preprocessor directives and statements of the host language are intermixed. When reading source code, many #ifdef and #endif directives distract from the actual code and can destroy the code layout (with *cpp*, every directive must be placed on its own line). There are cases in which preprocessor directives entirely obfuscate the source code as illustrated in Figure 16[11] and in our previous FemtoOS example in Figure 2. Furthermore, nested preprocessor directives and multiple directives belonging to different features as in Figure 1 are other typical causes of obfuscated code.

While language-based mechanisms such as feature-oriented programming avoid this obfuscation by separating feature code, researchers have explored several ways to improve the representation in the realm of preprocessors: First, textual annotations with a less verbose syntax that can be used within a single line could help, and can be used with many tools. Second, views can help programmers to focus on the relevant code, as discussed above. Third, visual means can be used to differentiate annotations from source code: Like some IDEs for PHP use different font styles or background colors to emphasize the difference between HTML and PHP in a single file, different graphical means can be used to distinguish

[11] In the example in Figure 16, preprocessor directives are used for Java code at a fine granularity [50], annotating not only statements but also parameters and part of expressions. We need to add eight additional lines just for preprocessor directives. Together with additional necessary line breaks, we need 21 instead of 9 lines for this code fragment.

preprocessor directives from the remaining source code. Finally, it is possible to eliminate textual annotations altogether and use the representation layer to convey annotations, as we show next.

In our tool CIDE, we abandoned textual annotations in favor of background colors to represent annotations [50]. For example, all code belonging to feature TRANSACTION is highlighted with background color red. Using the representation layer, also our example from Figure 16 is much shorter as shown in Figure 17. The use of background colors mimics our initial steps to mark features on printouts with colored text markers and can easily be implemented since the background color is not yet used in most IDEs. Instead of background colors the tool Spotlight uses colored lines next to the source code [29]. Background colors and lines are especially helpful for long and nested annotations, which may otherwise be hard to track. We are aware of some potential problems of using colors (e.g., humans are only able to distinguish a certain number of colors), but still, there are many interesting possibilities to explore; for example, usually a few colors for the features a developer currently focuses on are sufficient. Recently, the tool *FeatureCommander* combined background colors, lines, and several further enhancements in a way that scales for product lines with several hundred features [35].

Despite all visual enhancements, there is one important lesson. Using preprocessors does not require modularity to be dropped at all, but rather frees programmers from the burden of forcing them to physically modularize everything. Typically, most of a feature's code will be still implemented mostly modularly, by a number of modules or classes, but additional statements for method invocations may be scattered in the remaining implementation as necessary. In most implementations, there are rarely annotations from more than two or three features on a single page of code [47].

4.5 Summary

There are many directions from which we can improve annotation-based implementations without replacing them with alternative implementation approaches, such as feature-oriented programming. Disciplined annotations remove many low-level problems and open the implementation for further analysis; views emulate modularity by providing a virtual separation of concerns; and visualizations reduce the code cluttering. At the same time, we keep the flexibility and simplicity of preprocessors: Developers still just mark and optionally remove code fragments from a common implementation.

Together, these improvements can turn traditional preprocessors into a viable alternative to composition-based approaches, such as feature-oriented programming. Still there are trade-offs: For example, virtual separation does not support true modularity and corresponding benefits such as separate compilation, whereas compositional approaches have problems at a fine granularity. Even combining the two approaches may yield additional benefits. We have explored these differences and synergies elsewhere [47, 48]. Recently, we have explored also automated

transformations between the two representations [51]. We cannot make a recommendation for one or the other approach. We believe that much (empirical) evaluation is still necessary. Currently, we are exploring both paths in parallel.

Tooling. Basic preprocessors are widely available for most languages. For Java, Antenna is a good choice for which also tool integration in Eclipse and NetBeans is available. Most advanced concepts discussed here have been implemented in our tool CIDE as an Eclipse plugin.[12] CIDE uses the feature-model editor and reasoning engine from FeatureIDE. CIDE is open source and comes with a number of examples and a video tutorial. Visualizations have been explored further in *View Infinity*[13] and *FeatureCommander*,[14] the latter of which comes with Xenomai (a realtime extension for Linux with 700 features) as example. For graphical models, *FeatureMapper*[15] provides similar functionality.

5 Variability-aware analysis

The analysis of product lines is difficult. The exponential explosion (up to 2^n variants for n features) makes a brute-force approach infeasible. At the same time, checking only sampled variants or variants currently shipped to customers leads to the effect that errors can lurk in the system for a long time. Errors are detected late, only when a specific feature combination is requested for the first time (when the problem is more expensive to find and fix). While this may work for in-house development with only a few products per year (e.g., software bundled with a hardware product line), especially in systems in which users can freely select features (e.g., Linux), checking variants in isolation obviously does not scale.

Variability-aware analysis is the idea to lift an analysis mechanism for a single system to the product-line world. Variability-aware analysis extends traditional analysis by reasoning about variability. Hence, instead of checking variants, variability is checked locally where it occurs inside the product-line implementation (without variant generation). Variability-aware analysis has been proposed for many different kinds of analysis, including type checking [5, 53, 92], model checking [12, 27, 60, 76], theorem proving [95], and parsing [56]; other kinds of analyses can probably be lifted similarly. There are very different strategies, but the key idea is usually similar. We will illustrate variability-aware analysis with type checking, first for annotation-based implementations, then for composition-based ones. Subsequently, we survey different general strategies.

5.1 Type checking annotation-based implementations

To illustrate variability-aware type checking, we use the trivial hello-world program with three features shown in Figure 18: From this program, we can generate

[12] `http://fosd.net/cide`
[13] `http://fosd.net/vi`
[14] `http://fosd.net/fc`
[15] `http://featuremapper.org/`

```
1  #include <stdio.h>
2
3  #ifdef WORLD
4  char *msg = "Hello World\n";
5  #endif
6  #ifdef BYE
7  char *msg = "Bye bye!\n";
8  #endif
9
10 main() {
11 #if defined(SLOW) && defined(WORLD)
12   sleep(10);
13 #endif
14
15   println(msg);
16 }
```

Fig. 18. Hello-world example with annotations.

eight different variants (with any combination of WORLD, BYE, and SLOW). Quite obviously, some of these programs are incorrect: Selecting neither WORLD nor BYE leads to a dangling variable access in the println parameter (msg has not been declared); selecting both WORLD and BYE leads to a variable declared twice.

To detect these errors with a brute-force approach, we would simply generate and type check all eight variants individually. While brute force seems acceptable in this example, it clearly does not scale for implementations with many features. Instead, variability-aware type checking uses a lifted type system that takes variability into account.

As a first step, we need to reason about conditions under which certain code fragments are included. Czarnecki and Pietroszek coined them *presence conditions*, to describe the conditions under which a code fragment is included with a propositional formula (the code line is included iff the presence condition of that line evaluates to true) [31]. In our example, the formulas are trivial: WORLD for Line 4, BYE for Line 7, $SLOW \wedge WORLD$ for Line 12, and true for all other lines. With more complex #ifdef conditions and nesting, the formulas become more complex as described in detail elsewhere [83].

Now, we can formulate type rules based on presence conditions. For example, whenever we find an access to a local variable, we need to make sure that we can *reach* at least one declaration. In our example, we require that the presence condition of accessing msg (i.e., true) implies the presence condition of either declaration of msg (i.e., WORLD and BYE): $true \Rightarrow (WORLD \vee BYE)$. Since this formula is not a tautology, we detect that a variant selecting neither feature is not type correct. Similar reachability conditions for function calls are straightforward and uninteresting, because the target declaration in a header file has presence condition true. As an additional check, we require that multiple definitions with the same name must be mutually exclusive: $\neg(WORLD \wedge BYE)$. This check reports an error for variants with both features. If the product line has a feature model describing the valid variants, we are only interested in errors in valid

variants. By using a representation of the feature model as propositional formula $\mathtt{fm}$ (translations are straightforward, cf. Sec. 2.2), we check only variants that are valid with respect to the feature model: $\mathtt{fm} \Rightarrow (\mathtt{true} \Rightarrow (\mathit{WORLD} \vee \mathit{BYE}))$ and $\mathtt{fm} \Rightarrow \neg(\mathit{WORLD} \wedge \mathit{BYE})$ as illustrated in Figure 19.

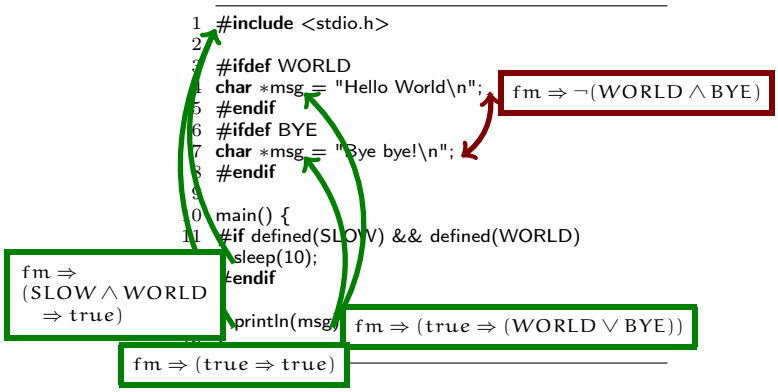

Fig. 19. Constraints in the hello-world example.

Abstracting from the example, we can define generic reachability and uniqueness conditions. A reachability condition between a caller and multiple targets is:

$$\mathtt{fm} \Rightarrow \mathtt{pc}(\mathtt{caller}) \Rightarrow \bigvee_{\mathtt{t} \in \mathtt{targets}} \mathtt{pc}(\mathtt{t})$$

where $\mathtt{pc}$ denotes a presence condition. The uniqueness condition that enforces that no variant defines multiple definitions is:

$$\mathtt{fm} \Rightarrow \bigwedge_{\mathtt{d}_1 \in \mathtt{definitions},\ \mathtt{d}_2 \in \mathtt{definitions},\ \mathtt{d}_1 \neq \mathtt{d}_2} \neg\big(\mathtt{pc}(\mathtt{d}_1) \wedge \mathtt{pc}(\mathtt{d}_2)\big)$$

Even for complex presence conditions and feature models, we can check whether these constraints hold efficiently with SAT solvers (Thaker et al. provide a good description of how to encode and implement this [92]).[16]

So, how does variability-aware type checking improve over the brute-force approach? Instead of just checking reachability and unique definitions in a single variant, we formulate conditions over the space of all variants. The important benefit of this approach is that we check variability locally, where it occurs. In our example, we do not need to check the combinations of SLOW and BYE, which are simply not relevant for typing. Technically, variability-aware

[16] Other logics and other solvers are possible, but SAT solvers seem to provide a sweet spot between performance and expressiveness [67].

type checking requires lookup functions to return *all possible* targets and their presence conditions. Furthermore, we might need to check alternative types of a variable. Still, in large systems, we do not check the surface complexity of 2^n variants, but analyze the source code more closely to find essential complexity, where variability actually matters. We cannot always avoid exponential blowup, but practical source code is usually well behaved and has comparably little local variability. Also, caching of SAT-solver queries is a viable optimization lever. Furthermore, the reduction to SAT problems enables efficient reasoning in practice, even in the presence of complex presence conditions and large feature models [53, 67, 92].

In prior work, we have described variability-aware type checking in more detail and with more realistic examples; we have formalized the type system and proven it sound (when the type system judges a product line as well-typed all variants are well-typed); and we have provided experience from practice [53].

5.2 Type checking composition-based implementations

The same concept of introducing variability into type checking can also be applied to feature-oriented programming. To that end, we first need to define a type system for our new language (as, for example, FFJ [6]) and then make it variability-aware by introducing reachability checks (as, for example, FFJ_{PL} [5]).

Since the type-checking mechanisms are conceptually similar for annotation-based and composition-based product lines, we restrict our explanation to a simple example of an object store with two basic implementations (example from [93]) that each can be extended with a feature AccessControl in Figure 20. Lookup of function calls works across feature boundaries and checking presence conditions is reduced to checking relationships between features.

More interestingly, the separation of features into distinct modules allows us to check some constraints *within a feature*. Whereas the previous approaches assume a closed world in which all features are known, separation of features encourages modular type checking in an open world. As illustrated in Figure 21, we can perform checks regarding fragments that are local to the feature. At the same time, we derive interfaces, which specify the constraints that have to be checked against other features. To check constraints between features, we can use brute force (check on composition) or just another variability-aware mechanism.

Modular type checking paves the road to true feature modularity, in which we distinguish between the public interface of a feature and private hidden implementations. Modular analysis of a feature reduces analysis effort, because we need to check each feature's internals only once and need to check only interfaces against interfaces of other features (checking interfaces usually is much faster than checking the entire implementation). Furthermore, we might be able to establish guarantees about features, without knowing all other features (open-world reasoning). For an instantiation of modular type checking of features, see the work on gDeep [3] and delta-oriented programming [81]. Li et al. explored a similar strategy for model checking [63].

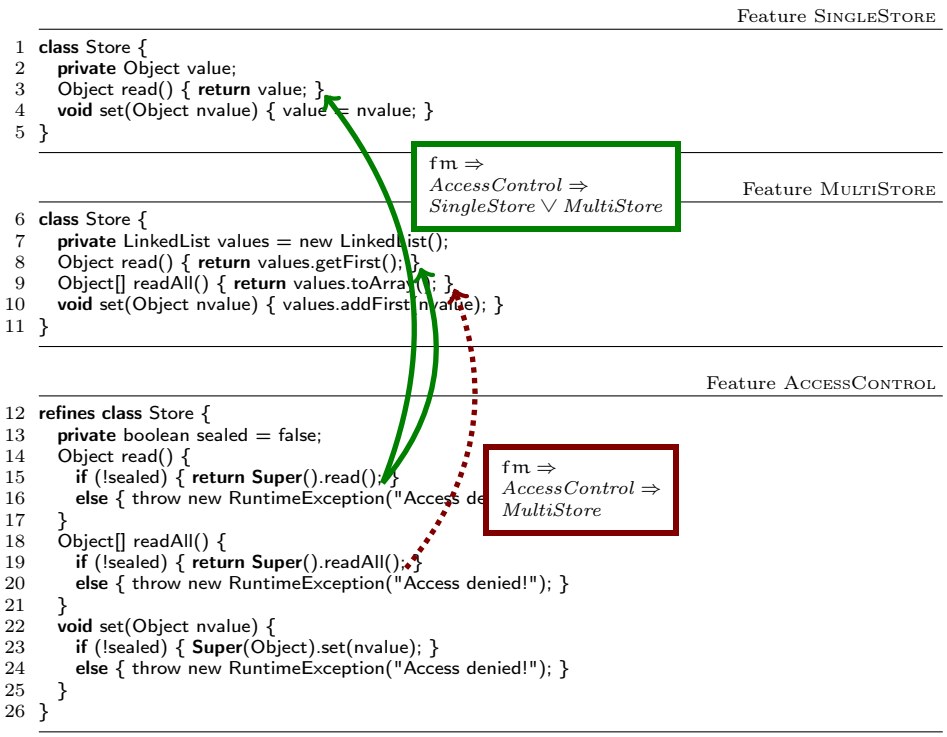

Fig. 20. Checking whether references to read and readAll are well-typed in *all* valid products.

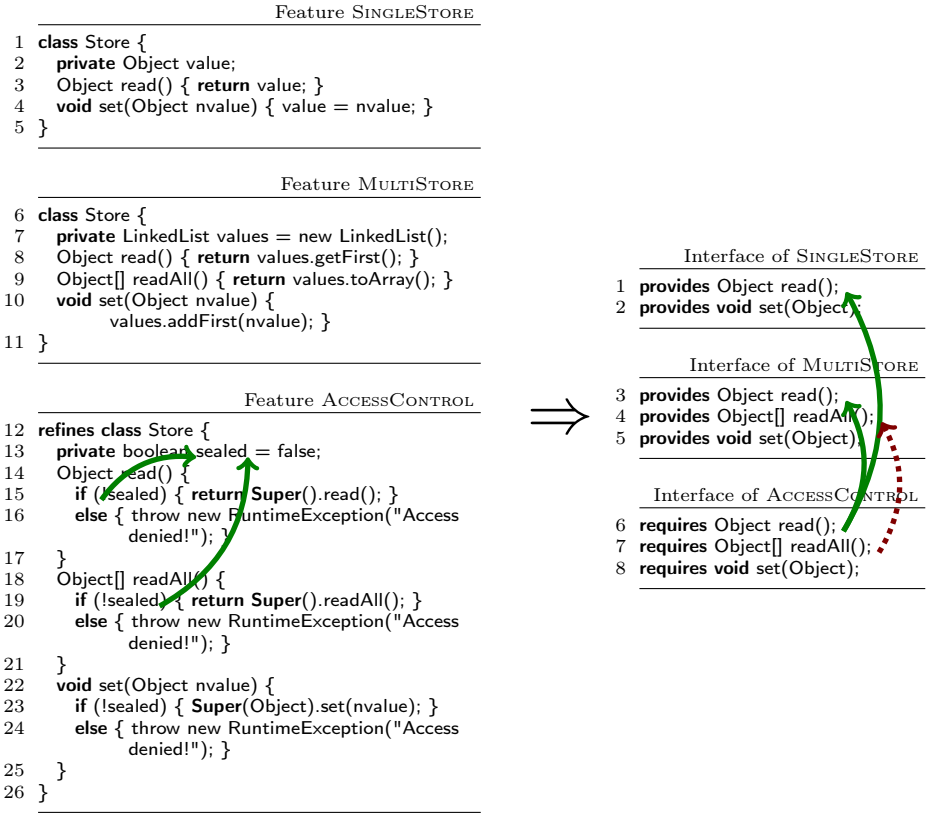

Fig. 21. References to field sealed can be checked entirely within feature AccessControl (left); references to read and readAll cut across feature boundaries and are checked at composition time based on the features' interfaces (right).

5.3 Analysis strategies

In general, we see three different strategies of how we can approach variability-aware analysis:

- *Brute-force strategy.* We check variants individually with standard analysis techniques. We can try to reduce effort by sampling relevant variants and focusing on certain coverage heuristics. For example, pair-wise feature coverage samples a small number of variants in the hope to discover all problems related to the interaction of pairs of features [72]. Especially for testing and measurement, approaches to select suitable variants have been explored [59, 72, 75, 82].
- *Family-based strategy.* We check the whole product line at once, as outlined for type checking above. We assume a closed world in which we know the implementation of all features and their relationships. The family-based strategy has been explored extensively for type checking and model checking [5, 12, 27, 31, 53, 60, 76, 92].
- *Feature-based strategy.* We check each feature in isolation as far as possible. Modular feature checks do not require implementation details of other features. For noncompositional properties that cannot be checked locally, we derive interfaces or constraints that must be checked when composing two features (per variant, or using a brute-force or family-based strategy). Modular checks avoid re-performing certain checks for each variant that are local to individual features; the strategy is suited especially if features are already separated. It has been explored, for example, for type checking, model checking and verification [3, 63, 81, 95].

These strategies can be applied to different forms of implementation and different kinds of analysis. Of course the strategies can be combined. For details on these strategies, their combinations, and a survey of existing analysis techniques see the recent report by Thüm et al. [93].

Tooling. Most variability-aware analyses, we are aware of, are in the state of research prototypes. See the corresponding references for further information. Our environment for virtual separation of concerns, CIDE, contains a variability-aware type system that covers large parts of Java. The *safegen* tool implements part of a variability-aware type system for the feature-oriented language Jak and is available as part of the AHEAD tool suite. We are currently in the process of integrating such type system into the *Fuji* compiler for feature-oriented programming in Java,[17] and afterward into FeatureIDE, and we are developing a type system for C code with #ifdefs as part of the TypeChef project.[18]

6 Open challenges

So far, we have illustrated different strategies to implement features in product lines. They all encourage disciplined implementations, that alleviate many prob-

[17] `http://fosd.net/fuji`
[18] `https://github.com/ckaestne/TypeChef`

lems traditionally associated with product-line implementations. Nevertheless, there are many open challenges.

A core challenge is the exponential explosion of the number of variants. The more features a product line supports, the more complex interaction patterns can occur that challenge maintenance tasks and quality assurance tasks. Although we have outlined possible strategies for variability-aware analysis, they cannot (yet) fully replace sophisticated software testing methods known from single-program development.

Feature interactions are especially problematic. A feature interaction occurs when two features behave different combined than they behave in isolation. A standard example are two features *flood control* and *fire alarm* in home-automation software that work well in isolation, but when combined, flood control may accidentally turn of sprinklers activated when a fire was detected [61]. When feature interactions are known, there are several implementation strategies, for example with additional derivative modules or nested preprocessor directives [55]. However, feature interactions can be difficult to detect, specify, and check against. Calder et al. provide a deeper introduction into the topic [25]. Many problems in product lines are caused by feature interactions.

Furthermore, both feature-oriented programming and preprocessor-based implementations have been criticized for neglecting modularity and overly relying on structures of the implementation. Although feature modules localize all feature code, only few approaches provide explicit interfaces that could enforce information hiding. We discuss this issue in detail elsewhere [52].

In general, also FOSD requires *variability management* as an essential task of project management. Developers should not add features, just because they can. Variability should always serve a mean for the project, such as answering to customer demands for tailor-made products, serving to a broader market segment, or preparing for potential customers. Variability adds effort, complexity, and costs for development, maintenance, and quality assurance. If (compile-time) variability is not really needed, it might be best to develop a traditional single program and use conventional development and testing approaches. However, if variability adds value to the project, as discussed in Section 2, the disciplined implementation approaches of FOSD discussed in this tutorial may provide a good balance between gained variability and required effort and costs.

7 Conclusion

With this tutorial, we have introduced FOSD. Beginning with basic concepts from the field of software product line engineering, we have introduced two approaches to FOSD: feature-oriented programming à la AHEAD and FeatureHouse and virtual separation of concerns. Subsequently, we have introduced the subfield of variability-aware analysis, which highlights a promising avenues of further work. We have covered only the basic concepts and a few methods, tools, and techniques, with a focus on techniques that can be readily explored. For further information,

we recommend a recent survey, which covers also related areas including feature interactions, feature design, optimization, and FOSD theories [4, 49].

Acknowledgements. Kästner's work is supported by the European Research Council, grant #203099 'ScalPL'. Apel's work is supported by the German DFG grants AP 206/2, AP 206/4, and LE 912/13.

References

1. B. Adams, B. Van Rompaey, C. Gibbs, and Y. Coady. Aspect mining in the presence of the C preprocessor. In *Proc. AOSD Workshop on Linking Aspect Technology and Evolution (LATE)*, pages 1–6. ACM Press, 2008.
2. F. I. Anfurrutia, O. Diaz, and S. Trujillo. On refining XML artifacts. In *Int'l Conf. Web Engineering*, volume 4607 of *Lecture Notes in Computer Science*, pages 473–478. Springer-Verlag, 2007.
3. S. Apel and D. Hutchins. A calculus for uniform feature composition. *ACM Trans. Program. Lang. Syst. (TOPLAS)*, 32(5):1–33, 2010.
4. S. Apel and C. Kästner. An overview of feature-oriented software development. *J. Object Technology (JOT)*, 8(5):49–84, 2009.
5. S. Apel, C. Kästner, A. Größlinger, and C. Lengauer. Type safety for feature-oriented product lines. *Automated Software Engineering*, 17(3):251–300, 2010.
6. S. Apel, C. Kästner, and C. Lengauer. Feature Featherweight Java: A calculus for feature-oriented programming and stepwise refinement. In *Proc. Int'l Conf. Generative Programming and Component Engineering (GPCE)*, pages 101–112. ACM Press, 2008.
7. S. Apel, C. Kästner, and C. Lengauer. FeatureHouse: Language-independent, automated software composition. In *Proc. Int'l Conf. Software Engineering (ICSE)*, pages 221–231. IEEE Computer Society, 2009.
8. S. Apel, S. Kolesnikov, J. Liebig, C. Kästner, M. Kuhlemann, and T. Leich. Access control in feature-oriented programming. *Science of Computer Programming (Special Issue on Feature-Oriented Software Development)*, 77(3):174–187, Mar. 2012.
9. S. Apel, T. Leich, M. Rosenmüller, and G. Saake. FeatureC++: On the symbiosis of feature-oriented and aspect-oriented programming. In *Proc. Int'l Conf. Generative Programming and Component Engineering (GPCE)*, volume 3676 of *Lecture Notes in Computer Science*, pages 125–140. Springer-Verlag, 2005.
10. S. Apel, T. Leich, and G. Saake. Aspectual feature modules. *IEEE Trans. Softw. Eng. (TSE)*, 34(2):162–180, 2008.
11. S. Apel and C. Lengauer. Superimposition: A language-independent approach to software composition. In *Proc. ETAPS Int'l Symposium on Software Composition*, number 4954 in Lecture Notes in Computer Science, pages 20–35. Springer-Verlag, 2008.
12. S. Apel, H. Speidel, P. Wendler, A. von Rhein, and D. Beyer. Detection of feature interactions using feature-aware verification. In *Proc. Int'l Conf. Automated Software Engineering (ASE)*, pages 372–375. IEEE Computer Society, 2011.
13. D. L. Atkins, T. Ball, T. L. Graves, and A. Mockus. Using version control data to evaluate the impact of software tools: A case study of the Version Editor. *IEEE Trans. Softw. Eng. (TSE)*, 28(7):625–637, 2002.
14. L. Bass, P. Clements, and R. Kazman. *Software Architecture in Practice.* Addison-Wesley, Boston, MA, 1998.

15. D. Batory. Feature models, grammars, and propositional formulas. In *Proc. Int'l Software Product Line Conference (SPLC)*, volume 3714 of *Lecture Notes in Computer Science*, pages 7–20. Springer-Verlag, 2005.
16. D. Batory, J. N. Sarvela, and A. Rauschmayer. Scaling step-wise refinement. *IEEE Trans. Softw. Eng. (TSE)*, 30(6):355–371, 2004.
17. I. Baxter and M. Mehlich. Preprocessor conditional removal by simple partial evaluation. In *Proc. Working Conf. Reverse Engineering (WCRE)*, pages 281–290. IEEE Computer Society, 2001.
18. D. Benavides, S. Seguraa, and A. Ruiz-Cortés. Automated analysis of feature models 20 years later: A literature review. *Information Systems*, 35(6):615–636, 2010.
19. D. Beuche, H. Papajewski, and W. Schröder-Preikschat. Variability management with feature models. *Sci. Comput. Program.*, 53(3):333–352, 2004.
20. V. Bono, A. Patel, and V. Shmatikov. A core calculus of classes and mixins. In *Proc. Europ. Conf. Object-Oriented Programming (ECOOP)*, volume 1628 of *Lecture Notes in Computer Science*, pages 43–66. Springer-Verlag, 1999.
21. J. Bosch. Super-imposition: A component adaptation technique. *Information and Software Technology (IST)*, 41(5):257–273, 1999.
22. L. Bouge and N. Francez. A compositional approach to superimposition. In *Proc. Symp. Principles of Programming Languages (POPL)*, pages 240–249. ACM Press, 1988.
23. C. Brabrand and M. I. Schwartzbach. Growing languages with metamorphic syntax macros. In *Proc. Workshop on Partial Evaluation and Semantics-Based Program Manipulation (PEPM)*, pages 31–40. ACM Press, 2002.
24. G. Bracha and W. Cook. Mixin-based inheritance. In *Proc. Int'l Conf. Object-Oriented Programming, Systems, Languages and Applications (OOPSLA)*, pages 303–311. ACM Press, 1990.
25. M. Calder, M. Kolberg, E. H. Magill, and S. Reiff-Marganiec. Feature interaction: A critical review and considered forecast. *Computer Networks*, 41(1):115–141, 2003.
26. L. Chen, M. A. Babar, and N. Ali. Variability management in software product lines: A systematic review. In *Proc. Int'l Software Product Line Conference (SPLC)*, pages 81–90. Carnegie Mellon University, 2009.
27. A. Classen, P. Heymans, P.-Y. Schobbens, A. Legay, and J.-F. Raskin. Model checking lots of systems: Efficient verification of temporal properties in software product lines. In *Proc. Int'l Conf. Software Engineering (ICSE)*, pages 335–344. ACM Press, 2010.
28. P. Clements and L. Northrop. *Software Product Lines: Practices and Patterns*. Addison-Wesley, Boston, MA, 2001.
29. D. Coppit, R. Painter, and M. Revelle. Spotlight: A prototype tool for software plans. In *Proc. Int'l Conf. Software Engineering (ICSE)*, pages 754–757. IEEE Computer Society, 2007.
30. K. Czarnecki and U. Eisenecker. *Generative Programming: Methods, Tools, and Applications*. ACM Press/Addison-Wesley, New York, 2000.
31. K. Czarnecki and K. Pietroszek. Verifying feature-based model templates against well-formedness OCL constraints. In *Proc. Int'l Conf. Generative Programming and Component Engineering (GPCE)*, pages 211–220. ACM Press, 2006.
32. M. Ernst, G. Badros, and D. Notkin. An empirical analysis of C preprocessor use. *IEEE Trans. Softw. Eng. (TSE)*, 28(12):1146–1170, 2002.
33. M. Erwig and E. Walkingshaw. The choice calculus: A representation for software variation. *ACM Trans. Softw. Eng. Methodol. (TOSEM)*, 21(1):6:1–6:27, 2011.

34. J.-M. Favre. Understanding-in-the-large. In *Proc. Int'l Workshop on Program Comprehension*, page 29. IEEE Computer Society, 1997.
35. J. Feigenspan, M. Schulze, M. Papendieck, C. Kästner, R. Dachselt, V. Köppen, and M. Frisch. Using background colors to support program comprehension in software product lines. In *Proc. Int'l Conf. Evaluation and Assessment in Software Engineering (EASE)*, pages 66–75, 2011.
36. R. E. Filman, T. Elrad, S. Clarke, and M. Aksit, editors. Addison-Wesley, Boston, MA, 2005.
37. R. Findler and M. Flatt. Modular object-oriented programming with units and mixins. In *Proc. Int'l Conf. Functional Programming (ICFP)*, pages 94–104. ACM Press, 1998.
38. M. Flatt, S. Krishnamurthi, and M. Felleisen. Classes and mixins. In *Proc. Symp. Principles of Programming Languages (POPL)*, pages 171–183. ACM Press, 1998.
39. A. Garrido. *Program Refactoring in the Presence of Preprocessor Directives*. PhD thesis, University of Illinois at Urbana-Champaign, 2005.
40. M. L. Griss, J. Favaro, and M. d' Alessandro. Integrating feature modeling with the RSEB. In *Proc. Int'l Conf. Software Reuse (ICSR)*, page 76. IEEE Computer Society, 1998.
41. S. Günther and S. Sunkle. Feature-oriented programming with Ruby. In *Proc. GPCE Workshop on Feature-Oriented Software Development (FOSD)*, pages 11–18. ACM Press, 2009.
42. A. N. Habermann, L. Flon, and L. Cooprider. Modularization and hierarchy in a family of operating systems. *Commun. ACM*, 19(5):266–272, 1976.
43. F. Heidenreich, I. Şavga, and C. Wende. On controlled visualisations in software product line engineering. In *Proc. SPLC Workshop on Visualization in Software Product Line Engineering (ViSPLE)*, pages 303–313. Lero, 2008.
44. D. Janzen and K. De Volder. Programming with crosscutting effective views. In *Proc. Europ. Conf. Object-Oriented Programming (ECOOP)*, volume 3086 of *Lecture Notes in Computer Science*, pages 195–218. Springer-Verlag, 2004.
45. I. John and M. Eisenbarth. A decade of scoping – a survey. In *Proc. Int'l Software Product Line Conference (SPLC)*, pages 31–40. Carnegie Mellon University, 2009.
46. K. Kang, S. G. Cohen, J. A. Hess, W. E. Novak, and A. S. Peterson. Feature-Oriented Domain Analysis (FODA) Feasibility Study. Technical Report CMU/SEI-90-TR-21, SEI, Pittsburgh, PA, 1990.
47. C. Kästner. *Virtual Separation of Concerns*. PhD thesis, University of Magdeburg, 2010.
48. C. Kästner and S. Apel. Integrating compositional and annotative approaches for product line engineering. In *Proc. GPCE Workshop on Modularization, Composition and Generative Techniques for Product Line Engineering*, pages 35–40. University of Passau, 2008.
49. C. Kästner and S. Apel. Virtual separation of concerns – A second chance for preprocessors. *Journal of Object Technology (JOT)*, 8(6):59–78, 2009.
50. C. Kästner, S. Apel, and M. Kuhlemann. Granularity in software product lines. In *Proc. Int'l Conf. Software Engineering (ICSE)*, pages 311–320. ACM Press, 2008.
51. C. Kästner, S. Apel, and M. Kuhlemann. A model of refactoring physically and virtually separated features. In *Proc. Int'l Conf. Generative Programming and Component Engineering (GPCE)*, pages 157–166. ACM Press, 2009.
52. C. Kästner, S. Apel, and K. Ostermann. The road to feature modularity? In *Proceedings of the Third Workshop on Feature-Oriented Software Development (FOSD)*, pages 5:1–5:8. ACM Press, Sept. 2011.

53. C. Kästner, S. Apel, T. Thüm, and G. Saake. Type checking annotation-based product lines. *ACM Trans. Softw. Eng. Methodol. (TOSEM)*, 21(3):14:1–14:39, 2012.
54. C. Kästner, S. Apel, S. Trujillo, M. Kuhlemann, and D. Batory. Guaranteeing syntactic correctness for all product line variants: A language-independent approach. In *Proc. Int'l Conf. Objects, Models, Components, Patterns (TOOLS EUROPE)*, volume 33 of *Lecture Notes in Business Information Processing*, pages 175–194. Springer-Verlag, 2009.
55. C. Kästner, S. Apel, S. S. ur Rahman, M. Rosenmüller, D. Batory, and G. Saake. On the impact of the optional feature problem: Analysis and case studies. In *Proc. Int'l Software Product Line Conference (SPLC)*, pages 181–190. Carnegie Mellon University, 2009.
56. C. Kästner, P. G. Giarrusso, T. Rendel, S. Erdweg, K. Ostermann, and T. Berger. Variability-aware parsing in the presence of lexical macros and conditional compilation. In *Proc. Int'l Conf. Object-Oriented Programming, Systems, Languages and Applications (OOPSLA)*, pages 805–824. ACM Press, Oct. 2011.
57. C. Kästner, T. Thüm, G. Saake, J. Feigenspan, T. Leich, F. Wielgorz, and S. Apel. FeatureIDE: Tool framework for feature-oriented software development. In *Proc. Int'l Conf. Software Engineering (ICSE)*, pages 611–614. IEEE Computer Society, 2009.
58. C. Kästner, S. Trujillo, and S. Apel. Visualizing software product line variabilities in source code. In *Proc. SPLC Workshop on Visualization in Software Product Line Engineering (ViSPLE)*, pages 303–313. Lero, 2008.
59. C. H. P. Kim, D. S. Batory, and S. Khurshid. Reducing combinatorics in testing product lines. In *Proc. Int'l Conf. Aspect-Oriented Software Development (AOSD)*, pages 57–68. ACM Press, 2011.
60. K. Lauenroth, K. Pohl, and S. Toehning. Model checking of domain artifacts in product line engineering. In *Proc. Int'l Conf. Automated Software Engineering (ASE)*, pages 269–280. IEEE Computer Society, 2009.
61. J. Lee, K. C. Kang, and S. Kim. A feature-based approach to product line production planning. In *Proc. Int'l Software Product Line Conference (SPLC)*, pages 183–196, 2004.
62. T. Leich, S. Apel, and L. Marnitz. Tool support for feature-oriented software development: FeatureIDE: An eclipse-based approach. In *Proc. OOPSLA Workshop on Eclipse Technology eXchange (ETX)*, pages 55–59. ACM Press, 2005.
63. H. C. Li, S. Krishnamurthi, and K. Fisler. Modular verification of open features using three-valued model checking. *Automated Software Engineering*, 12(3):349–382, July 2005.
64. J. Liebig, C. Kästner, and S. Apel. Analyzing the discipline of preprocessor annotations in 30 million lines of C code. In *Proc. Int'l Conf. Aspect-Oriented Software Development (AOSD)*, pages 191–202. ACM Press, 2011.
65. J. Liu, D. Batory, and C. Lengauer. Feature oriented refactoring of legacy applications. In *Proc. Int'l Conf. Software Engineering (ICSE)*, pages 112–121. ACM Press, 2006.
66. B. McCloskey and E. Brewer. ASTEC: A new approach to refactoring C. In *Proc. Europ. Software Engineering Conf./Foundations of Software Engineering (ESEC/FSE)*, pages 21–30. ACM Press, 2005.
67. M. Mendonça, A. Wąsowski, and K. Czarnecki. SAT-based analysis of feature models is easy. In *Proc. Int'l Software Product Line Conference (SPLC)*, pages 231–240. Carnegie Mellon University, 2009.

68. M. Mendonça, A. Wąsowski, K. Czarnecki, and D. D. Cowan. Efficient compilation techniques for large scale feature models. In *Proc. Int'l Conf. Generative Programming and Component Engineering (GPCE)*, pages 13–22. ACM Press, 2008.
69. T. Mens. A state-of-the-art survey on software merging. *IEEE Trans. Softw. Eng. (TSE)*, 28(5):449–462, 2002.
70. H. Ossher and W. Harrison. Combination of inheritance hierarchies. In *Proc. Int'l Conf. Object-Oriented Programming, Systems, Languages and Applications (OOPSLA)*, pages 25–40. ACM Press, 1992.
71. H. Ossher and P. Tarr. Hyper/J: Multi-dimensional separation of concerns for Java. In *Proc. Int'l Conf. Software Engineering (ICSE)*, pages 734–737. ACM Press, 2000.
72. S. Oster, F. Markert, and P. Ritter. Automated incremental pairwise testing of software product lines. In *Proc. Int'l Software Product Line Conference (SPLC)*, volume 6287 of *Lecture Notes in Computer Science*, pages 196–210. Springer-Verlag, 2010.
73. Y. Padioleau. Parsing C/C++ code without pre-processing. In *Proc. Int'l Conf. Compiler Construction (CC)*, pages 109–125. Springer-Verlag, 2009.
74. D. L. Parnas. On the design and development of program families. *IEEE Trans. Softw. Eng. (TSE)*, 2(1):1–9, 1976.
75. K. Pohl, G. Böckle, and F. J. van der Linden. *Software Product Line Engineering: Foundations, Principles and Techniques.* Springer-Verlag, Berlin/Heidelberg, 2005.
76. H. Post and C. Sinz. Configuration lifting: Verification meets software configuration. In *Proc. Int'l Conf. Automated Software Engineering (ASE)*, pages 347–350. IEEE Computer Society, 2008.
77. C. Prehofer. Feature-oriented programming: A fresh look at objects. In *Proc. Europ. Conf. Object-Oriented Programming (ECOOP)*, volume 1241 of *Lecture Notes in Computer Science*, pages 419–443. Springer-Verlag, 1997.
78. R. Rabiser, P. Grünbacher, and D. Dhungana. Supporting product derivation by adapting and augmenting variability models. In *Proc. Int'l Software Product Line Conference (SPLC)*, pages 141–150. IEEE Computer Society, 2007.
79. J. G. Refstrup. Adapting to change: Architecture, processes and tools: A closer look at HP's experience in evolving the Owen software product line. In *Proc. Int'l Software Product Line Conference (SPLC)*, 2009. Keynote presentation.
80. M. Rosenmüller, S. Apel, T. Leich, and G. Saake. Tailor-made data management for embedded systems: A case study on Berkeley DB. *Data and Knowledge Engineering (DKE)*, 68(12):1493–1512, 2009.
81. I. Schaefer, L. Bettini, and F. Damiani. Compositional type-checking for delta-oriented programming. In *Proc. Int'l Conf. Aspect-Oriented Software Development (AOSD)*, pages 43–56. ACM Press, 2011.
82. N. Siegmund, M. Rosenmüller, M. Kuhlemann, C. Kästner, S. Apel, and G. Saake. SPL Conqueror: Toward optimization of non-functional properties in software product lines. *Software Quality Journal - Special issue on Quality Engineering for Software Product Lines*, 2012. in press.
83. J. Sincero, R. Tartler, D. Lohmann, and W. Schröder-Preikschat. Efficient extraction and analysis of preprocessor-based variability. In *Proc. Int'l Conf. Generative Programming and Component Engineering (GPCE)*, pages 33–42. ACM Press, 2010.
84. N. Singh, C. Gibbs, and Y. Coady. C-CLR: A tool for navigating highly configurable system software. In *Proc. AOSD Workshop on Aspects, Components, and Patterns for Infrastructure Software (ACP4IS)*, page 9. ACM Press, 2007.

85. Y. Smaragdakis and D. Batory. Mixin layers: An object-oriented implementation technique for refinements and collaboration-based designs. *ACM Trans. Softw. Eng. Methodol. (TOSEM)*, 11(2):215–255, 2002.
86. H. Spencer and G. Collyer. #ifdef considered harmful or portability experience with C news. In *Proc. USENIX Conf.*, pages 185–198. USENIX Association, 1992.
87. M. Steger, C. Tischer, B. Boss, A. Müller, O. Pertler, W. Stolz, and S. Ferber. Introducing PLA at Bosch Gasoline Systems: Experiences and practices. In *Proc. Int'l Software Product Line Conference (SPLC)*, volume 3154 of *Lecture Notes in Computer Science*, pages 34–50. Springer-Verlag, 2004.
88. C. Szyperski. *Component Software: Beyond Object-Oriented Programming.* Addison-Wesley, Boston, MA, 2nd edition, 2002.
89. P. Tarr, H. Ossher, W. Harrison, and S. M. Sutton, Jr. N degrees of separation: Multi-dimensional separation of concerns. In *Proc. Int'l Conf. Software Engineering (ICSE)*, pages 107–119. IEEE Computer Society, 1999.
90. R. Tartler, D. Lohmann, J. Sincero, and W. Schröder-Preikschat. Feature consistency in compile-time-configurable system software: Facing the Linux 10,000 feature problem. In *Proc. European Conference on Computer Systems (EuroSys)*, pages 47–60. ACM Press, 2011.
91. A. Tešanović, K. Sheng, and J. Hansson. Application-tailored database systems: A case of aspects in an embedded database. In *Proc. Int'l Database Engineering and Applications Symposium*, pages 291–301. IEEE Computer Society, 2004.
92. S. Thaker, D. Batory, D. Kitchin, and W. Cook. Safe composition of product lines. In *Proc. Int'l Conf. Generative Programming and Component Engineering (GPCE)*, pages 95–104. ACM Press, 2007.
93. T. Thüm, S. Apel, C. Kästner, M. Kuhlemann, I. Schaefer, and G. Saake. Analysis strategies for software product lines. Technical Report FIN-004-2012, School of Computer Science, University of Magdeburg, Apr. 2012.
94. T. Thüm, D. Batory, and C. Kästner. Reasoning about edits to feature models. In *Proc. Int'l Conf. Software Engineering (ICSE)*, pages 254–264. IEEE Computer Society, 2009.
95. T. Thüm, I. Schaefer, M. Kuhlemann, and S. Apel. Proof composition for deductive verification of software product lines. In *Proc. Int'l Workshop on Variability-Intensive Systems Testing, Validation & Verification (VAST)*, pages 270–277. IEEE Computer Society, 2011.
96. M. Vittek. Refactoring browser with preprocessor. In *Proc. European Conf. on Software Maintenance and Reengineering (CSMR)*, pages 101–110. IEEE Computer Society, 2003.
97. D. Weise and R. Crew. Programmable syntax macros. In *Proc. Conf. Programming Language Design and Implementation (PLDI)*, pages 156–165. ACM Press, 1993.
98. N. Wirth. Program development by stepwise refinement. *Commun. ACM*, 14(4):221–227, 1971.