
Nr.: FIN-017-2009

# Representing and Composing First-class Features with FeatureJ

Sagar Sunkle, Sebastian Günther, Gunter Saake

*Database Research Group*


Fakultät für Informatik
Otto-von-Guericke-Universität Magdeburg

Technical report

Nr.: FIN-017-2009

# Representing and Composing First-class Features with FeatureJ

Sagar Sunkle, Sebastian Günther, Gunter Saake

*Database Research Group*

Technical report (Internet)
Elektronische Zeitschriftenreihe
der Fakultät für Informatik
der Otto-von-Guericke-Universität Magdeburg
ISSN 1869-5078


Fakultät für Informatik
Otto-von-Guericke-Universität Magdeburg

**Impressum** (§ 5 TMG)

*Herausgeber:*
Otto-von-Guericke-Universität Magdeburg
Fakultät für Informatik
Der Dekan

*Verantwortlich für diese Ausgabe:*
Otto-von-Guericke-Universität Magdeburg
Fakultät für Informatik
Sagar Sunkle
Postfach 4120
39016 Magdeburg
E-Mail: sagar.sunkle@iti.cs.uni-magdeburg.de

http://www.cs.uni-magdeburg.de/Technical_reports.html

Technical report (Internet)
ISSN 1869-5078

*Redaktionsschluss*: 27.11.2009

*Bezug:* Otto-von-Guericke-Universität Magdeburg
Fakultät für Informatik
Dekanat

# Representing and Composing First-class Features with FeatureJ

Sagar Sunkle, Sebastian Günther, and Gunter Saake

School of Computer Science, University of Magdeburg,
39106 Magdeburg, Germany
sagar.sunkle, sebastian.guenther,
gunter.saake@iti.cs.uni-magdeburg.de

**Abstract.** Software product lines (SPLs) enable creating product families, set of products that differ in terms of features. Traditionally, techniques for implementing features have sided with one of the two views of features: as distinguishable characteristics or as increments/changes in program functionality of the software under consideration. We argue that in order to realize the full potential of features as a separation of concerns mechanism or a modularity mechanism in its own right, both of these views must be supported by the representation of features. Furthermore, the composition of such uniformly represented features should be streamlined so that both can evolve together. Towards this end, we present FeatureJ, an implementation technique that integrates features, variants, and product lines as first-class entities, namely types, in the Java programming language. We review the trends in the representation and composition of features in the current implementation techniques and arrive at a set of requirements to represent features as first-class entities. We demonstrate the syntax of FeatureJ and explain its compiler architecture with a running example of a product line. We compare our implementation approach with other approaches in terms of the representation and composition of features and state the advantages that our approach brings to the implementation of SPLs.

## 1 Introduction

*Features* are the main reuse mechanism of Software Product Lines (SPLs). SPLs represent a software engineering paradigm that enables creating a set of products, called product families, based on the common and the variable features. Feature-oriented software development (FOSD) is a general term applied to techniques (henceforth called *feature approaches*) used to synthesize products of a product family [1]. Since the introduction of the approaches such as feature-oriented programming, there has been a proliferation of techniques that attempt to implement features [6,7,16,29,31,33,36]. As the concepts related to software product lines evolve, many of these technique prove inadequate 1) to provide basic representation of features that capture the new requirements of representing the evolving SPL concepts and 2) to provide new mechanisms of composition that effectively leverage such representations [40].

In our position paper [40], we put forward a representation of features that would address various problems related to features such as forced hierarchical refinements, limitation of ordered composition, inexpressiveness of program deltas, type support for features, and dynamic composition of features. We showed how the new representation can be used to alleviate these problems. After presenting our ideas regarding a new representation for features in [40], we set out to implement them in a software called FeatureJ, an extension of Java. We have implemented a number of small case studies in FeatureJ such as the Graph Product Line (GPL), the Notepad Product Line (NPL), and the Expression Product Line (EPL)[1]. We have also implemented many examples indicating various other capabilities of FeatureJ such as implementing multiple product lines and multiple variants. Currently we are exploring ways to extends FeatureJ in terms of both the representation and composition capabilities of features. We intend to implement larger case studies such as refactoring large open-source software like OpenJDK[2] to a feature-based version to investigate the composition of language features, previously refactored Berkeley DB[3] to make comparative analysis of FeatureJ's capabilities with other approaches in which Berkeley DB has been implemented[4], and other applications such as Google Web Toolkit[5] which allows developers to create Javascript front-end applications in Java, to study cross-language application of features, etc.

In this paper, we take a review of how features are represented in FeatureJ and how it composes the features thus represented. We begin by describing what features are and how we distinguish them based on whether the features being referred to are the features at the analysis level, or they are the actual features being implemented. We specify why the themes of feature representation and feature composition are important themes which must be studied in concert. We

[1] http://firstclassfeatures.org/index.php?n=Examples.FeatureJ
[2] http://openjdk.java.net/
[3] http://www.oracle.com/database/berkeley-db/je/index.html
[4] http://wwwiti.cs.uni-magdeburg.de/iti_db/research/cide/
[5] http://code.google.com/webtoolkit/

review the trends in the representation of features in various existing feature approaches, followed by how the features are composed in each of those approaches. Based on this analysis, we obtain a set of requirements which we translate into design choices for implementing FeatureJ. We explain the syntax and semantics of typed representation of features in FeatureJ by means of a running example of the EPL. We elaborate on the FeatureJ compiler architecture that enables the new representation of features envisioned in [40]. We describe how features are represented as types and how the typed features are composed as an integral part of compilation process in FeatureJ. We discuss how FeatureJ differs from the other feature approaches based on the categories of feature representation and composition explained earlier and state the advantages of FeatureJ. We conclude this paper with the discussion of further extensions possible with respect to the representation and the composition of features in FeatureJ.

## 2 What are features?

A feature of a software system represents a particular functionality provided by that system. FOSD uses features to denote the requirements of a stakeholder. A specific software product is created by selecting certain features from an available set of features. In FOSD, features are modeled using the feature diagrams which graphically show the relationships between the features [11,25]. We first take review of feature diagrams and elaborate on the general theme of features introducing a couple of terms that distinguish between kinds of features.

### 2.1 Feature Models

SPLs are modeled using the feature diagrams. We explain feature diagrams using the NPL[6] as shown in Figure 1. The NPL is based on the idea that the variants of a notepad application can be obtained by selecting specific features such as the creating, editing, and formatting options. The NPL itself is the root feature. The NPL has four top level features: three mandatory features - File, Edit, and Format, and one optional feature - Help. These features have the *and* relation, which means that all of them can be selected together in a variant, except the feature Help which is optional. The NewDocument and the Open features have the *or* relation, which means that whenever a variant of the NPL contains the Create feature, it must also contain one or more of the features NewDocument and Open. The FontColor and the BGColor features of the feature Color have the *alternative* relation which means that when the feature Color is selected in a variant of the NPL, then either of the FontColor or BGColor features must be selected but not both.

There are two inclusion constraint in this feature diagram, showing arcs from the features Paste and FontColor to the features ClipBoard and Color respectively. This means that whenever the feature Paste is selected so must the feature

[6] http://firstclassfeatures.org/index.php?n=FeatureJ.NotepadPL

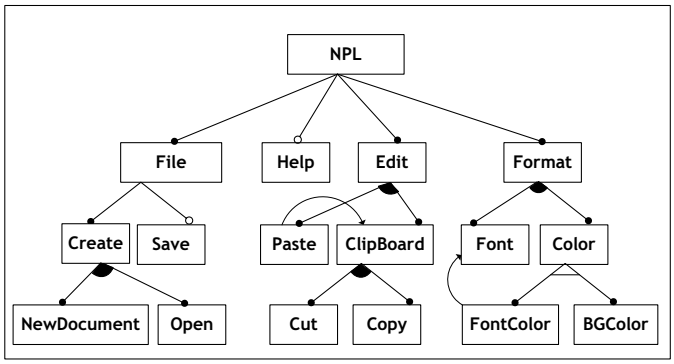



Fig. 1: Feature Model of the Notepad Product Line

ClipBoard, and whenever the feature FontColor is selected so must the feature Color. A notepad product variant is obtained by making selection of features starting with all the top level features and further down the feature diagram by following the feature relations for each parent feature. Based on the various relations and the constraint, a valid notepad product variant could be the set of features {File,Edit, Format, Help, Create, NewDocument, ClipBoard, Cut, Copy, Font, Color, BGColor}.

### 2.2 Conceptual and Concrete Features

In the rest of the paper, we differentiate between the features as 1) the *conceptual features* - the features in feature model / features in the analysis and design stages of the FOSD and 2) the *concrete features* - the implemented features (the nature of which differs based on the feature approach) / code fragments constituting conceptual features to be used in software product configuration and generation [21]. While the conceptual features are more or less uniformly represented or reasoned about [8,14,16,19], the representations of the concrete features are as plenty as there are feature approaches. This kind of distinction is necessary because as features at the analysis and conceptual levels, the features in the feature models merely denote functionalities and relationships between them, but have no special semantics attached to them [30]. On the contrary every feature approach attaches semantic meaning (to a varying degree based on the representation of features in it) to code fragments that constitute a concrete feature. Henceforth, we talk about features in models as the conceptual features and implemented features as the concrete features.

## 3 Feature Representation and Composition

A feature is an end-user-visible, distinguishable characteristic of a system that is relevant to a stakeholder of the system [11]. A feature is also defined to be an

increment in functionality [7]. In [40], we related the weaknesses in various feature approaches enlisted in [5,31,35] to the inadequate representations of features in those approaches. We proposed that these two definitions essentially point out two sides of the same coin, and as such both must be considered together in a given feature approach. In the next section, we specify the importance of the themes of feature representation and composition and why they should be considered in concert.

### 3.1 Importance of Feature Representation and Feature Composition

A feature as a domain concept was first introduced in the seminal work on feature-oriented domain analysis (FODA) by Kang et al.[25]. A feature as a software implementation paradigm was first put forward by Christian Prehofer in [37]. Neither of these two lines of thought are complete without each other. Domain concepts about a software system eventually need to be implemented. Similarly, feature approaches such as feature-oriented programming where features are used as a separation of concern mechanism [7], are incomplete as an SPL implementation technique without an overlying structure of the software system to which it is supposed to be applied. This structure is precisely the domain concepts of that software system [1]. In other words, in real software systems to be modeled as SPLs, the conceptual and concrete features need to have similar semantics (indicating a uniform feature representation mechanism) and the relationship between the conceptual features as well as their mappings to the concrete features, need to be streamlined (indicating a coherent feature composition mechanism). In the next two sections, we will give many examples of this dichotomy and relate it to the representation and composition themes for features.

### 3.2 Feature Representation

Feature representation in general has been treated lightly. It was considered that so far as it achieved its intent of adding/composing functionalities, any representation of features served well. We make this point clear by tracing different ways in which features have been represented in various existing feature approaches.

**3.2.1 Using preprocessors and annotations** Preprocessing techniques have been used to implement features, such as conditional compilation using #ifdef statements in C++, or Munge which uses preprocessor directives hidden inside comments in Java[7]. These techniques do not require the use of other concerns to implement features as they are generally part of the host language compiler. The code belonging to different features is simply put inside preprocessor directives, and added to the source to be compiled depending on the features selected. The use of pre-processor techniques for implementing features has been heavily criticized. It is found by many researchers that using preprocessors to implement

[7] http://weblogs.java.net/blog/tball/archive/2006/09/munge_swings_se.html

features - 1) neglects the principle of separation of concerns, 2) is sensitive to syntax, type, and behavioral errors, 3) obfuscates the source code, 4) and finally, severely limits the reuse [28] of features. Interestingly, in spite of the lack of abstraction for the conceptual and concrete features, preprocessing approaches are used in commercial software for product lines such as pure::variants [39], perhaps owing to the conceptual simplicity of using preprocessors for implementing features.

The virtual separation of concerns [28] is a unique approach that virtually overlays annotations on the program elements belonging to specific features. It is implemented on the top of the Eclipse IDE and is capable of using many code related features of Eclipse. It attempts to address the problems related to preprocessors by 1) using colors to annotate features virtually, 2) providing support for views on features thereby emulating modularization of features, and 3) using disciplined annotations to prevent syntax, type, and behavioral errors. Although the virtual separation of concerns approach is conceptually intuitive, it lacks some obvious advantages of feature modularization. The reuse of feature related code is still a problem in this approach since there is no way to address the concrete features programmatically. Also if a code fragment belongs to multiple features it is colored as many times, making it difficult to distinguish between features. Although the editable views alleviate this complexity to some extent, the need for more sophisticated tool support to address these issues becomes apparent in complex scenarios [28].

**3.2.2 Using concern-specific modularization mechanisms to implement features** This category of feature approaches basically uses a representation of some other concern, such as e.g., aspects, teams, hyper-slices, units, and traits etc., to implement features. What this means, is that main modularization mechanism used in the underlying implementation technology is anything but features. The rationale is to use the ability of given feature approach to select/modify functionality of a program even though feature is not the main mechanism of separation of concern in that technology. Examples of this kind are implementing features in AspectJ via aspects [29], in Object Teams with teams and roles [23], in Jiazzi via atoms and units [31], in Scala via traits [31], and in HyperJ via hyper-slices [31]. Kästner et al. [29] implemented a case study to refactor the Berkley DB to a feature-based application with 38 features. They used aspects to represent features in terms of classes, inner classes, method extensions, and so on, essentially the elements in the Berkley DB source code that were part of different features. They found that 1) the code readability and maintainability decreased along with the number of features and correspondingly with the number of aspects that were used to represent these features, 2) all refactoring to feature-based design had to be done manually (because there was no explicit representation of feature models), and 3) most expressive and powerful constructs in AspectJ were not used (most of the unique characteristics of aspects become unimportant when implementing features so long as functionality can be introduced).

Hundt et al. presented a case study implementing features using Object Teams [23]. In teams, concrete features must be mapped to teams, the main modularization mechanism in the team programming model. A team can be treated like a class as well as like a package that groups classes. Roles are classes contained in teams that can be used to decorate other classes. Crosscutting features are implemented as aspects. There are at least 5 different modular entities, namely, classes, packages, teams, roles, aspects etc. Developers need a deep familiarity with the teams programming model, the sheer number of different modular entities interacting with each other creates a complex application scenario which is difficult to maintain and evolve.

Lopez-Herrejon et al. [31] implemented features using Jiazzi components [34] called units: atoms built from Java programs and compounds built from atoms. Units are the main modularization mechanism and the main implementation concern. Units have different modularization structure than classes. Different features have to be packaged properly across atoms and compounds. Lopez-Herrejon et al. [31] note that defining features in terms of Jiazzi signatures which describe the packaging of code related to the concrete features and expressing the relationships between the conceptual features was a non-trivial task.

They also implemented features in the Scala programming language using traits [17], a unit of modularization in Scala that represents an abstract class without state. They found that implementing program deltas, i.e. the code components of features, in terms of traits was a non-trivial task as well, especially specifying the order of method extensions with which program deltas were manipulated.

In the same case study, Lopez-Herrejon et al. [31] also implemented features in HyperJ [36]. HyperJ represents concerns at a level of abstraction higher than the individual concerns [36]. Its an implementation of multidimensional separation of concerns (MDSOC). A software system in HyperJ is implemented with the idea that it can contain multiple concern dimensions (classes and features or classes and aspects), and that it can be executed by projecting individual programs of the system by projecting their execution along concern hyperplanes. Features can be implemented as hyperslices that group all units implementing the feature. We believe that although the idea of higher abstraction in MDSOC is appealing, there are many drawbacks when the treatment of a individual concern such as features is concerned. For example, Chitchyan et al. [10] found that 1) in HyperJ, primary requirements, i.e., features, can not be traced in the composed system, 2) it is difficult to comprehend how the composed system is affected when addressing additional requirements.

**3.2.3 Using structural forms of feature representation** In this category of feature approaches, we review those that use some form of representation for features themselves as opposed to using other concerns. The AHEAD tool suite [7] and FeatureC++ [4] are examples of such approaches. Both the AHEAD tool suite and FeatureC++ treat features as refinements. The program variants are created by composing features (refinements to base classes, stored in specific

folders) with the base program. Base classes and refinements (classes with the same names but containing feature related code) are arranged in a hierarchy of folders on a disk. A folder with only the refinements to specific base classes denotes a feature. Both AHEAD and FeatureC++ have been used successfully in many case studies. Although they are one step higher in abstraction to represent features, they have been known to face many problems [31,35]. We call this type of representation of features as the structural representation, because features now have their own representation (instead of being represented with other concerns) and this representation is encapsulated in structures external to the programs. This requires some kind of composer that would compose base programs and refinements from different folders specified in equation files [9]. Apel et al. [3,6] present a language independent superimposition approach to feature composition. They introduce a new programming model called feature structure tree (FST) which represents the abstract hierarchical structure of software components. Features of software components represented such as base programs and features as folders in AHEAD[7] are composed by composing the feature structure trees. An FST is a high level abstraction of hierarchical representation (such as folders on a disk or XML documents with tags that act as containers). This approach makes FSTs a very useful and more general software composition approach but as far as features are considered, one form of structural representation (folders) is replaced with another (FSTs).

These representations are one step closer to representing features natively. What they miss out is a connection to feature modeling concepts. AHEAD programs are composed using equation files that contain equations which determine the order of composition of refinements in different folders. An equation file is essentially a rudimentary form of the modeling specification. In the next category of feature approaches, we review approaches that consider the important relation between feature modeling concepts and actual implementation of features. Although these approaches do not represent features themselves at the most appropriate level of abstraction, they provide support for feature modeling and provide a link between feature modeling and feature implementation.

**3.2.4 Using representation of feature modeling concepts towards feature implementations** Deursen and Klint proposed a feature description language in [16]. Although their research is more domain specific language (DSL) oriented than related to feature implementations, they created a DSL over feature diagrams called Feature Description Language (FDL) which is of our interest. The FDL represents conceptual features in terms of feature definitions and operations. Feature operations in FDL are constructed using feature diagram algebra. This algebra consists of various rules such as transformation of the feature expressions to disjunctive normal formulas, normalizing these transformation to enumerate possible variants etc. The constraints in the feature diagrams are converted into satisfiability constraints over the propositional formulas. A variant is represented as a list of selected features. The concrete features are represented by classes. The relations between conceptual features are modeled with aggrega-

tion, associations, subclasses, etc [16]. This indicates that the concrete features have no explicit representation in the FDL.

Loughran et al. presented the variability modeling language (VML) in [33]. The VML supports first-class representation of architectural variabilities. It is an architectural description language; it considers the problem space to be feature models whereas architectural models are the solution space. While the VML descriptions of the conceptual features consist of representation of variation points, cardinalities of features, constraints, etc., the concrete features are implemented as components.

This category of approaches indicates the need of a good representation, not only for the concrete features, but also the conceptual features which are in essence execution specifications for a given product line.

#### Summarizing feature representations

We summarize the theme of feature representation in the following:

- There has been a progression from using preprocessor mechanism with no abstraction for features, to using a complex non-native representation of features (because they are represented in terms of other concerns), to giving them a structural representation with the composition specification external to the implementation, to using a representation of feature models to guide the process of product derivation.
- In any feature representation, two components are required to be addressed - one, the actual implementation of features themselves and two, the structure of features, variants, and the product lines, expressed in terms of feature models, i.e., the features and the composition specification, respectively. Any representation addressing only one of the two would fall short of achieving full capabilities of features as semantic entities of their own accord.
- Only when both the conceptual and concrete features are represented as same entities, the process of composing them could be streamlined.

In the next section, we describe the theme of feature composition with respect to the four categories of feature representation. Although some details of feature composition are already laid out in the description of feature representations, we try to abstract the progression of feature composition methods used along with these representations.

### 3.3 Feature Composition

Feature composition is the process of combining together various fragments of code scattered in different kinds of modules implementing a software product line to obtain an executable version of a product variant. Feature composition is intricately connected to feature representation. The ways in which features in given approach can be composed depend substantially upon how they are represented: whether using preprocessing mechanisms, whether in terms of other

concerns, externally in a structural format, or deriving from model specifications. We make this point clear by taking review of how composition is carried out with respect to each of the aforementioned categories of feature representation.

**3.3.1 Composing features using preprocessors and annotations** The composition of features represented using preprocessors and annotations happens via conditional compilation and code generation based on views respectively. Since there is a complete lack of abstraction (at least in case of preprocessors), it is extremely difficult to realize a reusable form of composition or configuration specification.

Previously, feature approaches have been segregated into compositional and annotative approaches [26]. Obtaining a product variant using preprocessors or annotations consists of *projecting* the selected features along with the base program. We deem this to be *feature composition* rather than *feature projection* (which we deem the process of obtaining only the code fragments belonging to features as opposed to a complete variant which would also contain the base program), since according to our definition, feature composition is any process in which a product variant is synthesized from a set of features.

**3.3.2 Composing features represented as other concerns** Implementing features in terms of teams in Object Teams calls for complex modular arrangement of feature related code [23]. Object Teams maps features to constructs that control activation of teams and roles (classes contained in teams). This mapping of feature models to teams and roles must follow structural semantics of Object Teams e.g., different kinds of features such mandatory, optional, alternative, exclusive features with is-a or part-of relations must be mapped properly to individual teams and roles forming a hierarchy of teams. Composition of features thus represented happens via activation teams, teams that control activation of other teams [23]. Similar observations can be made about features implemented in terms of Jiazzi units or Scala traits. Features represented as Jiazzi units must follow the modular hierarchy of atoms and units. Composition specification for these atoms and units is in the form of the open class pattern, which is used to define feature composition. When represented as traits in Scala, features must be represented in terms of combinations of Scala classes and traits and the composition specification needs to be applied in terms of deep mixin composition [31]. When features are represented in terms of other concerns and corresponding modular forms, the composition of such features depends on the semantics of these concerns and their modular entities. The semantics of concerns might be such that the developer might have to follow workarounds (to create appropriate mappings) or it might be more than enough capable of implementing various feature related functionality ( such as when using aspects to implement features; most semantic capabilities of aspects remain unused [29]). Furthermore, features implemented in code must have a composition specification of one form or the other. In other words, although features represented as other concerns can in-

deed be composed because modular entities of that concern can be composed, the developer still needs further configuration specification.

**3.3.3 Composing features represented using structural forms** Features are represented in AHEAD in terms of refinements, which are placed in specific folders. An equation file dictates the order of composition of these refinements. The refinements are weaved using aspectual mechanisms. FeatureC++ goes one step further and provides support for dynamic composition via code generation by creating a delegation hierarchy atop refinement classes [38]. This further complicates the feature composition because to use dynamic composition SPL classes and the delegation classes must be arranged in a proper hierarchy. In case of composition with FSTs by Apel et al. [6], components (any modular structures) are represented as tree data structure. Two trees are composed by adding contents of the root node and from there on proceeding recursively. This composition requires that these nodes are at the same level as well as their names and types are same. FSTs act at a higher level of abstraction, but still require composition specification (in terms of folder hierarchy in AHEAD for composition of AHEAD modules using FSTs).

In both these cases, a step toward native representation does not add more capability to features as native entities, because features are simply abstracted away in a structural form, but no implicit composition specification in the form representation for feature modeling concepts exists.

**3.3.4 Composing features represented with the feature modeling specification** Feature diagrams represented in the FDL are mapped to UML diagrams as discussed in Section 3.2.4. These mappings are represented using XML metadata information exchange format and imported to UML modeling tools which are used for generating Java classes [16]. The VML uses the concept of references and actions [33], which act as component activation and component composition decisions. Along with the feature descriptions, the references and actions are used to compose components to obtain product variants.

Both FDL and VML come close to our idea of representing feature modeling concepts as a language. While they score on this end of the line, the other end is left loose by not providing native representation for features actually implemented in the code. Using native representation of only the feature modeling concepts falls short in alleviating indirections one has to follow in creating product variants.

**Summarizing feature composition**

We summarize the theme of feature composition in the following:

- Feature composition is substantially influenced by the way features are represented.

– When the concrete features have no native representation, their composition translates into composition of other concerns, external structures, or mapping to components or code generation tools. This results in alienating the identity of both the conceptual and concrete features in the composed variants. Consequently, such problems as feature untraceability, difficulty in reconfigurations etc. arise.
– These problems can be avoided if both the conceptual and concrete features were represented uniformly, with their composition integrated in a common semantics.

In the next section, we specify the requirements of representing and composing features as first-class entities. We elaborate on FeatureJ, an extension of Java, with which we realize our vision of the features as first-class entities. We lay out the background with the explanation of the JastAdd extensible compiler system, followed by the class-loading idiosyncrasies in Java. We then describe in detail the syntax and semantics of FeatureJ, by discussing the FeatureJ compiler architecture.

## 4 FeatureJ

Based on the discussion so far, we can observe that either there are approaches that take a step towards giving the implemented features a better representation or those that have represented feature modeling concepts natively. Neither of these have a uniform representation of both the implemented features as well as the feature modeling concepts. This is the main motivation behind creating FeatureJ. It is one of our design goals that we match feature modeling with feature implementation in this language extension, i.e., represent the conceptual and concrete features by the same mechanism, which also includes composition of both kinds. Before we proceed to explain how features are represented and composed in FeatureJ, we outline some of the properties of the features represented natively or the features represented as first-class entities. With first-class features, the product line concepts in a given domain can be realized more clearly at the implementation level. Based on the properties of first-class entities, we specify the requirements of implementing features as first-class entities as enlisted below :

1. Instead of using other mechanisms to represent and compose features, features should be represented as native first-class types or first-class entities in the host language [40].
2. First-class features should represent the conceptual and concrete features uniformly as discussed in Section 3.2.
3. Semantics of the first-class features should be rich enough to subsume feature composition as discussed in Section 3.3.
4. The identity of features should be retained throughout the life cycle of an application to enable a more controlled manipulation of the functionality of features [40].

5. First-class features should be extensible with respect to advances in feature modeling and feature composition.

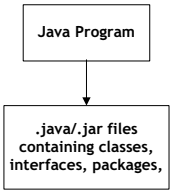
While we chose the Java programming language to be the host language as it is used prominently in product line research, the first-class representation of features outline above can be implemented in any programming language. As a matter of fact, we have implemented an initial prototype of the first-class features as a language extension of the Ruby language, called *rbFeatures* [20,21].

### 4.1 Java Classloading Idiosyncrasies

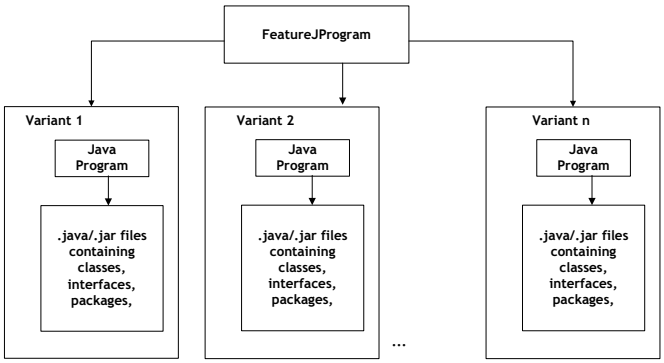
A Java application runs with a Java Virtual Machine (JVM) that interacts with Java classes in the .class file format. Creating first-class product lines and product variants atop a Java application indicates that for a given application many program variants may exists simultaneously. Consequently, we have to address different versions of the application classes. This is shown in Figure 2. Whereas a regular Java program only runs a single version of class(es)/interface(s) comprising it, running a FeatureJ program consists of running many versions of the same class(es)/interface(s) (belonging to different variants). Various properties of Java classes, class loaders, and classpaths come into play in such a scenario.

***Referring to different versions of the same class*** This scenario takes place when multiple variants exist simultaneously consisting of different versions of the same SPL class. All classes in a Java application are loaded using some subclass of the ClassLoader class. Classes are referred to by the qualified name format, which begins with the package name and ends with the class name. However, this qualified name is not sufficient for the JVM to distinguish a class. The JVM identifies a class uniquely by a combination its qualified name and the effective class loader, i.e., the class loader that loads this class. The loading of the user classes by the custom class loaders is delegated to the primordial class loader. When both versions of the class A are on the classpath, they are equivalent to the JVM because their qualified name is same and they share the same class loader which is the primordial class loader. To distinguish between the classes from different variants, they must be placed in a path that is not in the classpath of the SPL application.

***Reloading modified versions of the same class*** This scenario takes place when a variant is modified by adding or removing a feature. When a class is loaded, all classes it references are loaded recursively. A class is only loaded once and then cached in the class loader by the JVM to ensure that the byte code of the class does not change. Reloading the class is therefore not possible using Java's class loaders. To reload a class the developer needs to implement a custom class loader. Even with a custom class loader, classes that have been resolved and loaded cannot be reloaded by the same custom class loader. Also, the objects of classes that have exactly the same qualified name but loaded by two different ClassLoader instances are treated as objects from different classes.




(a) Java program




(b) FeatureJ program with many program variants

Fig. 2: Difference between the Java and FeatureJ program execution

Object of one of such classes cannot be cast to the other and doing so results in the ClassCastException because their classLoaders do not match. In order to overcome this limitation, two methods can be used. 1) The developer can use an interface as the object type and reload the implementing class, or 2) the developer can use a superclass as the object type and reload the extending class. In both these cases, the type of the object, i.e., the interface or the superclass is not reloaded. Keeping the interface or the superclass constant, we can reload the implementing class or the extending class respectively.

### 4.2 JastAdd: Extensible Compiler System

We use JastAdd[8],a Java based compiler construction system [22], to implement FeatureJ. The JastAdd compiler system enables modular and extensible specification of compiler analyses. JastAdd is based on a language formalism called Rewritable Circular Reference Attributed Grammars (ReCRAGs). It uses an

[8] http://jastadd.org/the-jastadd-extensible-java-compiler

object-oriented Abstract Syntax Tree as the main data structure. Various language entities are represented as nodes and sub-trees and semantic behavior is added in terms of attributes on such nodes. Behavioral modules are composed using static aspects. The ReCRAGs allow decoupled computation of various analyses, e.g., if type checking module needs information from name resolution module then such information is obtained by evaluating concerned attributes without having to explicitly specify the order. As the behavior specification is independent of the order of attribute evaluation (from the developer's perspective) different chunks of behavior can be added to language entities assuming that all the necessary information for this module is available. We chose JastAdd for two compelling reasons: 1) By delegating the task of obtaining specific piece of information to separate attributes (which are themselves grouped modularly in JastAdd aspects), JastAdd makes writing various compiler analyses extremely intuitive. 2) The JastAdd compiler contains implementations of Java 1.4 and 1.5 frontends and backends. This implementation is known to comply to the Java language specification more than other Java compiler implementations [18].

## 5 Feature Representation in FeatureJ

Based on the requirements of first-class feature representation and composition outlined in Section 4, we implemented features as first-class entities in the Java programming language. We made two design choices when implementing FeatureJ.

1. We implemented features as well as product lines and product variants as types, more precisely as reference types in Java. The reason for this is that considering a native representation there is no other language entity in Java that is semantically rich enough to capture everything we wanted to implement about features as first-class entities.
2. Given the fact we wanted to reify the features (which means that we wanted all the information about features, product lines, and product variants preserved and available at runtime), we had two options to begin with: 1) store them as bytecode attributes to be processed later or 2) reify the features to a meta-class library that is external to the AST but still part of the FeatureJ compiler. In the first case, it would have been difficult to make use of JastAdd's attributes based computational capabilities (as JastAdd contains a Java to bytecode backend but not the other way round). We therefore chose the second option.

Based on our first design choice, we extended the Java 1.4 AST with type declarations for product lines, product variants, and features, as well as its lexical and parsing specifications to accommodate the FeatureJ vocabulary. The reification of FeatureJ types takes place via the meta-classes PL, PLVariant, and Feature. This compiler design is shown in Figure 3. Different extensions to FeatureJ itself are modularized into a set of lexer, parser, AST, and semantics files which are combined together and composed with corresponding Java

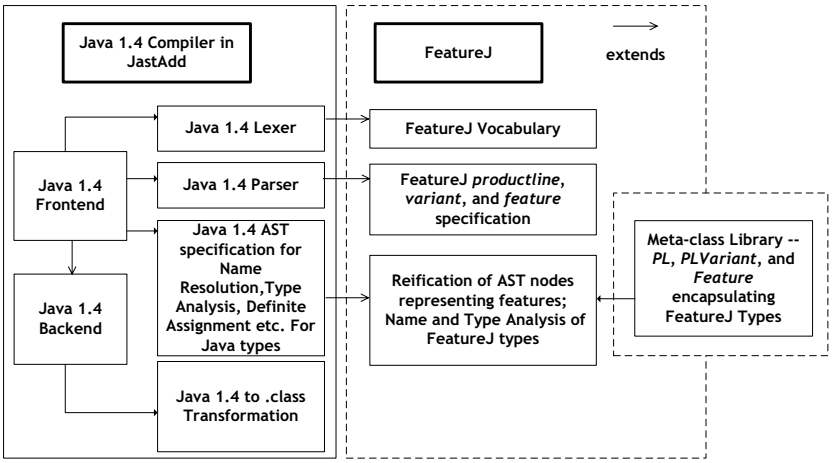



Fig. 3: FeatureJ Implementation using JastAdd

1.4 compiler specifications of JastAdd. Since the Java 1.5 compiler in JastAdd is itself an extension of Java 1.4 compiler, FeatureJ can be used as is for the language constructs in Java 1.5 that are common to Java 1.4. Furthermore, for newly added constructs (such as enhanced for statements), FeatureJ works without additional lexical, parsing, and AST components, because feature containments (blocks of code containing code fragments belonging to features) in FeatureJ include language entities at an abstract level, such as any statements in a method (including enhanced for statements which subclass the Statement class in the AST). Any extension built upon JastAdd's implementations of Java 1.4/1.5 compilers can be made *feature-aware* by combining that extension with FeatureJ. For completely new language constructs such as annotations in Java 1.5, FeatureJ can be extended with the minimum of effort to include them in feature containments.

### FeatureJ Syntax

In order to describe the FeatureJ syntax, we make use of the EPL. The EPL is an application of SPL concepts to the expression problem [32]. It treats the functionality of adding different operations on predefined data types as features. The data types consist of literals, additive expressions, negated expressions whereas the operations consist of printing and evaluating the expressions. The EPL can be represented as a two-dimensional matrix in which rows and columns indicate combinations of the data types and the operations [31]. We represent this matrix as a feature diagram as shown in Figure 4. The feature ap indicates the functionality of applying the printing operation on the additive expressions. Similarly, the feature ne indicates the functionality of applying the evaluation operation to

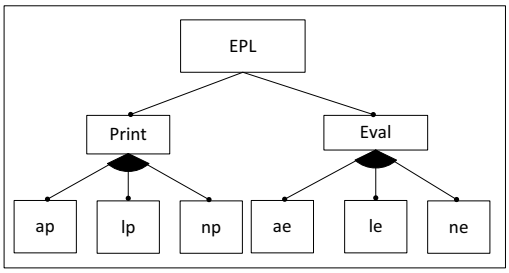



Fig. 4: Feature Model of the Expression Product Line

negated expressions. Listings 1.1 to 1.7 show the EPL implementation adapted from [31]. The FeatureJ implementation of the EPL[9] uses six classes (including the test driver and the application launcher) and one interface Exp. The interface Exp (Listing 1.2) contains the two methods print() and eval() corresponding to the two operations printing and evaluation. The classes Lit, Add, and Neg extend the interface Exp (Listings 1.3,1.4, and 1.5 respectively). The class EPLTest (Listing 1.6) is a test driver to test various combinations of data types and operations. The class PLDefine (Listing 1.1) contains the EPL declaration, whereas the class Launcher (1.7) contains declaration of two variants and shows how they can be executed in FeatureJ.

**Listing 1.1: The EPL declaration in FeatureJ**

```
1 package testEPL;
2 public class PLDefine {
3   productline EPL {
4     features {
5       Print: more(lp, ap, np),
6       Eval : more(le, ae, ne)
7     }
8     all(Print, Eval)
9   };
10 }
```

***productline Declaration Syntax.*** Listing 1.1 shows the *productline* type declaration for the EPL. The *features* block in the *productline* type declaration of the EPL contains features expressions separated by commas. Each expression in the *features* block identifies a pair of a parent feature and the child feature(s) along with the quantifier (denoting the feature relation). The *more* quantifier on the feature set {lp,ap,np} indicates that in a given *variant* type definition, one or more of the features lp, ap, and ne can be selected for the parent feature Create. The top level features along with the top level quantifier are specified

[9] http://firstclassfeatures.org/index.php?n=FeatureJ.ExpressionPL

immediately outside the *features* block. The optionality of features is indicated by the ? symbol. The EPL does not contain any constraints. The constraints, if present, between a pairs of features are defined in the *constraints* block. The expression $A$ <-> $B$ is used to indicate an inclusion constraint between the features A and B and the expression $A$ >-< $B$ is used to indicate an exclusion constraint between the features A and B.

A *productline* type declaration is not required to be specified in a separate class. It can appear as a body declaration in any of the SPL classes since its scope is package wide.

***Feature Containments Syntax.*** Various code fragments belonging to the features declared in the *productline* type are put inside what we call *feature containments*. Listings 1.2 to 1.6 show different kinds of feature containments available in FeatureJ.

**Listing 1.2: Interface for expressions**

```
1 package testEPL;
2 public interface Exp {
3   feature EPL lp {
4     public void print();
5   }
6   feature EPL le {
7     public int eval();
8   }
9 }
```

The lines 9-13 and 14-18 in Listing 1.3 show feature containments of methods print() and eval() in the Lit class for features lp and le respectively. For a given class A, any or all of the field declarations, methods, constructors, and inner classes can be part of feature containments in FeatureJ. The lines 3-8 in Listing 1.3 show feature containment of the combination of a field declaration and the constructor of the Lit class.

**Listing 1.3: Implementing the literals**

```
1 package testEPL;
2 public class Lit implements Exp {
3   feature EPL lp, EPL le {
4     int value;
5     public Lit(int v) {
6       value = v;
7     }
8   }
9   feature EPL lp {
10    public void print() {
11      System.out.println(value);
12    }
13  }
14  feature EPL le {
```

```
15      public int eval() {
16        return value;
17      }
18    }
19  }
```

Listings 1.4 and 1.5 show similar feature containments in classes Add and Neg respectively.

**Listing 1.4: Implementing the additive expressions**

```
1  package testEPL;
2  public class Add implements Exp {
3    feature EPL ap, EPL ae {
4      Exp left, right;
5      Add(Exp l, Exp r) {
6        left = l;
7        right = r;
8      }
9    }
10   feature EPL ap {
11     public void print() {
12       left.print();
13       System.out.print("+");
14       right.print();
15       System.out.println();
16     }
17   }
18   feature EPL ae {
19     public int eval() {
20       return left.eval() + right.eval();
21     }
22   }
23 }
```

The code fragments constituting more than one features are contained by specifying all the features to which they belong as shown in the line 3 in each of the listings 1.4 and 1.5.

**Listing 1.5: Implementing the negative expressions**

```
1  package testEPL;
2  public class Neg implements Exp {
3    feature EPL np, EPL ne {
4      Exp expr;
5    public Neg(Exp e) {
6        expr = e;
7      }
8    }
9    feature EPL np {
10     public void print() {
```

```
11        System.out.print("-(");
12        expr.print();
13        System.out.print(")");
14      }
15    }
16    feature EPL ne {
17      public int eval() {
18        return expr.eval() * -1;
19      }
20    }
21  }
```

The lines 3, 4, and 5 in listing 1.6 show individual field declarations contained by multiple features. Like classes, any or all of the statements inside a method, a constructor, or an inner class can be part of the feature containments in FeatureJ. This is shown in lines 7-9, 10-12, and 13-15 in Listing 1.6, where field assignments statements inside the EPLTest constructor are contained by different features. Similarly, lines 18-23 in Listing 1.6 show individual statements (method calls for methods print() and eval()) contained in different features.

**Listing 1.6: Test driver for EPL**

```
1  package testEPL;
2  public class EPLTest {
3    feature EPL lp, EPL le { Lit ltree; }
4    feature EPL ap, EPL ae  { Add atree; }
5    feature EPL np, EPL ne { Neg ntree; }
6    public EPLTest() {
7      feature EPL lp, EPL le {
8        ltree = new Lit(3);
9      }
10     feature EPL ap, EPL ae {
11       atree = new Add(ltree, ltree);
12     }
13     feature EPL np, EPL ne {
14       ntree = new Neg(ltree);
15     }
16   }
17   public void run() {
18     feature EPL lp  { ltree.print();}
19     feature EPL ap { atree.print();}
20     feature EPL np { ntree.print();}
21     feature EPL le  { System.out.println(ltree.eval());}
22     feature EPL ae { System.out.println(atree.eval());}
23     feature EPL ne { System.out.println(ntree.eval());}
24   }
25 }
```

***Variant declaration and generation syntax.*** To obtain a specific variant of the EPL *productline*, a *variant* declaration is used as shown in lines 5-8 and

9-12 of Listing 1.7. The expression *variant EPL addExpression* indicates that addExpression is a *variant* of the EPL *productline*. The *variant* types addExpression and negateExpression in the Launcher class indicate two possible ways in which the productline type EPL can be configured.

The *productline, variant*, and *feature* types thus declared need to be bound or registered to their respective meta-classes as shown in Listing 1.7. We call the PL, PLVariant, and Feature classes as meta-classes in the sense that these classes are used by the FeatureJ compiler to govern the behavior of the corresponding FeatureJ types. To process the FeatureJ types in a variety of ways, composing and modifying variants at runtime, querying the structures of the *productline* and *variant* types, and querying various properties of the feature types, such as its dependent features etc. can be done using instances of the corresponding meta-classes.[10] A PL meta-class object encapsulates the *productline* type specified in its constructor's argument as shown in the line 14 of Listing 1.7. Similarly, a PLVariant meta-class object encapsulates a *variant* type. The arguments to the constructor of a PLVariant object must also specify the PL object which encapsulates the parent *productline* type of this *variant* type as shown in lines 16 and 21 of Listing 1.7. The objects of the classes specific to a given *variant* type can be accessed by using the getVariantObject method of a PLVariant object as shown in lines 17-18 and 22-23 of Listing 1.7.

**Listing 1.7: Binding types to meta-classes and executing variants**

```
1 package testEPL;
2 import com.unimag.sag.featurej.*;
3 import com.unimag.sag.featurej.meta.*;
4 public class Launcher  {
5   variant EPL addExpression {
6     Print  = [lp and ap],
7     Eval = [le and ae]
8   };
9   variant EPL negateExpression {
10    Print  = [lp and np],
```

[10] These meta-classes are conceptually similar to the meta-classes in the Groovy language[24]. Groovy is a dynamic language for the JVM. The meta-classes in Groovy implement the meta-object protocol for the Groovy and Java classes and mechanisms such as aspect orientation can be built in Groovy by extending the meta-classes. This is related to our second design choice for FeatureJ. By using instances of the FeatureJ meta-classes to represent FeatureJ types, we can easily reify all properties of the FeatureJ types, without resorting to internal transformation of FeatureJ types to byte-code representation. This also enables a developer to process feature, variants, and product lines at compile time via the FeatureJ types, and at runtime, via the instances of the meta-classes. Although the use of these classes can be completely hidden behind a custom syntax for type instantiation and type usage for the FeatureJ types so that a developer only deals with the FeatureJ types, we have chosen to retain the use of these classes for the current version of FeatureJ for experimental purposes. In future versions of FeatureJ we intend to add such custom syntax for FeatureJ types.

```
11      Eval = [le and ne]
12    };
13    public static void main(String[] args) {
14      PL EPL=new PL("EPL");
15      //Running addExpression variant
16      PLVariant AExpr=new PLVariant(EPL,"addExpression");
17      EPLTest aExprTest=
18      (EPLTest)AExpr.getVariantObject("EPLTest");
19      aExprTest.run();
20      //Running negateExpression variant
21      PLVariant NExpr=new PLVariant(EPL,"negateExpression");
22      EPLTest nExprTest=
23      (EPLTest)NExpr.getVariantObject("EPLTest");
24      nExprTest.run();
25    }
26  }
```

When the *productline, variant*, and *feature* types are encapsulated inside the objects of the meta-classes PL, PLVariant, and Feature, they follow the visibility mechanisms applicable to regular Java classes[11]. On the other hand, the scope of both the *variant* and *feature* types is package wide and they are always used by qualifying their names with the name of the *productline* type to which they belong.

### Default granularity of features in FeatureJ

In FeatureJ, the default granularity of features is as follows:

- Inside a class, one or more class body declarations, i.e., field declarations, method declarations, and constructor declarations, can be contained in a feature. To indicate to FeatureJ that the entire class is to be part of a feature, all class body declarations need to be contained in that feature. Different body declarations within a class may belong to zero (indicating that the body declaration belongs to the base program) or more features. The same is applicable to inner classes.
- Inside an interface, one or more interface body declarations, i.e., abstract method declarations and constants, can be contained in a feature. Like a class, to indicate to FeatureJ that the entire interface is part of a feature, all interface body declarations need to be contained in that feature.
- Inside a method, one or more statements can be contained in a feature. Different statements or sets of statements may belong to different features. The same is applicable to constructors.

The default granularity of features in FeatureJ can be easily extended to finer level such as a *for* statement, so that the statements inside the for block can be

[11] See examples of encapsulating *feature* type in an object of meta-class Feature in the Graph Product Line example - http://firstclassfeatures.org/index.php?n=FeatureJ.GraphPL

contained in a feature. To extend the default granularity levels in FeatureJ, a developer needs to implement the ContainerNode and ConstituentNode interfaces in the FeatureJ compiler. An example of classes implementing the ContainerNode and ConstituentNode interfaces is the FeatureJ AST classes ClassDecl and FeatureAsMemberInClassesDecl respectively. To extend the granularity to *for* statements, a developer would have to create an AST node representing set of statements inside the *for* block, such as FeatureAsMemberInForStmt. Then the ForStmt class in the FeatureJ AST (representing *for* statements) needs to implement the ContainerNode interface whereas the FeatureAsMemberInForStmt would need to implement the ConstituentNode interface. The granularity can similarly be extended to a coarser level such as packages, so that one or more classes in a package can be contained in a feature.

## 6 Feature Composition in FeatureJ

FeatureJ represents both the conceptual and concrete features by the same mechanism, as types. This enables performing modeling operations as well as composition as part of type analysis process of FeatureJ. In the next section we explain the FeatureJ compiler architecture that makes this possible and then elaborate on the feature composition process.

### 6.1 FeatureJ Compiler Architecture Enabling First-class Features

As discussed earlier in Section 4.1, FeatureJ differs from other implementation approaches in the fact that multiple product lines and product variants may exist at the same time (this similar to multiple objects of a class existing simultaneously). Once various SPL concepts are raised to the first-class status, this is a step analogically similar to one class multiple objects scenario. This requires a meta-level access to the FeatureJ types which is provided via meta-classes PL, PLVariant, and Feature. These meta-classes not only oversee the binding of productline, variant, and feature types to PL, PLVariant and Feature objects respectively, but also provide the runtime access to these types. FeatureJ's compiler architecture that enables first-class representation and composition of features is shown in Figure 5. It consists of two phases: in the first phase, the infrastructure required for variant generation and execution is created and the second phase constitutes the composition of features.

***First phase - creating the infrastructure*** The FeatureJ compiler proceeds in two phases when input with .fjava files and the required .jar files. The first phase in the FeatureJ compiler consists of 1) parsing the .fjava files 2) creating the folder hierarchy for later generation of program variants, 3) generating interfaces for the SPL classes, and 4) enabling selection of execution type. The syntactic error(s) if any, is found during parsing of the .fjava files. If there is no syntactic error (in FeatureJ type declarations or feature containments) then the compilation proceeds to the first phase where a folder hierarchy is generated

that corresponds to the variant definitions and the packages in the classes of the SPL. For example, given that EPL classes are part of the testEPL package and the program contains two variant definitions (addExpression and negateExpression) then FeatureJ generates following directories in a temporary directory: 1) Main\testEPL - For storing the abstract classes corresponding to SPL classes + any interfaces in the testEPL package 2) addExpression\testEPL and negateExpression\testEPL - for storing the implementations of the abstract classes. These directories are used in the second phase to store implementations of SPL classes, thus providing a separate namespace. This enables avoiding the classpath problems stated earlier in Section 4.1. To avoid classloading problems in reloading different versions of classes from different variants, FeatureJ generates an abstract class for each SPL class, that provides a common interface against which specific implementations are generated based on the variant type definitions.

If there are many packages or nested packages in SPL classes then a corresponding hierarchy of folders is generated. Thus, given that one of the SPL classes is the class Neg 1.5, then an abstract class Neg is generated in Main\testEPL folder as shown in Listing 1.8, the implementations of which are generated in the addExpression\testEPL and negateExpression\testEPL folders in the second phase.

**Listing 1.8: Abstract class Neg for the SPL class Neg**

```
1 package testEPL;
2 public abstract class Neg extends Object implements Exp {
3   public abstract  int eval();
4   public abstract  void print();
5 }
```

FeatureJ preserves any inheritance relationships (via abstract classes or interfaces) that pre-exist between the classes in a FeatureJ program.

FeatureJ requires that the user specify an application entry point. In the final step of the first phase, an application container is generated using the specified application entry point based upon the execution type of FeatureJ program selected earlier, i.e, whether the FeatureJ program is to be run as a regular Java application or as a test application (this is done by selecting a -appMain or -testMain command line option).

***Second phase - variant generation using FeatureJ types*** The main idea behind FeatureJ is to reify the features and the related information and to raise the status of product lines, variants, and features to being first-class entities. A FeatureJ program may contain many *productline* types over a set of classes, interfaces, and packages, which are written as .fjava files with feature declarations and containments and sent to the FeatureJ compiler. The FeatureJ compiler processes each *productline* via the objects of the meta-class PL, and its variants are generated via the objects of the meta-class PLVariant. Each *variant* essentially contains specific code compositions of classes, interfaces, and packages that were input to the FeatureJ compiler. The user can interact with the variant programs

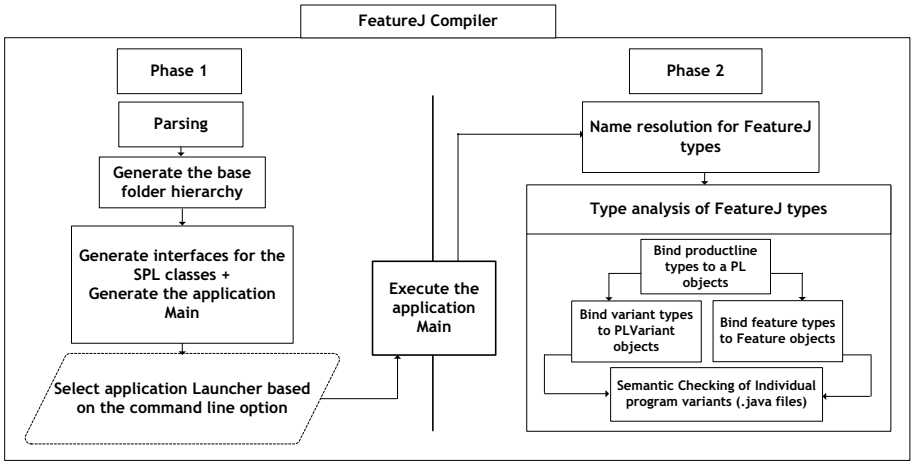



Fig. 5: FeatureJ Compiler Architecture

using the objects of classes of a specific *variant*. These objects are the objects of the regular Java classes and as such can be used as parameters to a method call, return values from functions, iterator objects (e.g., to iterate over only the leaf features of a variant) etc. We begin by describing the name resolution process of FeatureJ types and follow with the explanation of type analysis of the same in the next section.

### 6.2 FeatureJ Name Resolution

A FeatureJ program may contain many *productline* types and correspondingly many *variant* and *feature* types. The name resolution of these types builds upon the Java name resolution mechanism of JastAdd as discussed in Section 4.2. These types do not have the access level modifiers, because they all have package level visibility by default. The *feature* and *variant* types are declared within the context of the parent *productline* type. A *productline* type in a different package can be imported just like the regular Java types in that package. When resolving *variant* and *feature* type names, they are checked against their parent *productline* type. This typically consists of checking for missing/double declarations of a *feature* type, checking for existence of *feature* types used in the *variant* type declaration, checking for existence of *feature* type used in the feature containment expressions, and checking that a *variant* type of a given *productline* type refers only to the features of that *productline*.

### 6.3 FeatureJ Type Analysis

The type analysis in FeatureJ consists of steps ranging from validating variant types against parent *productline* type to binding all FeatureJ types to meta-class objects and composing individual variants.

***Structural validation of variant types.*** The first step consists of checking the structure of a variant based on its parent *productline* type for any inconsistencies in the feature selection based on the specified quantifiers and the inclusion and exclusion constraints. This basically involves validating the quantifiers on groups of features. For example, if the *productline* type consists of an expression like *Eval : more(le, ae, ne)*, then FeatureJ ensures that a variant of this *productline* must select at least one or more of the features le, ae, or ne for the *feature* Eval.

***Constraint resolution per variant type(s).*** In the second step FeatureJ checks that all inclusion and exclusion constraints are satisfied for a given variant, e.g., suppose there was an inclusion constraint like ae <-> ne in a *productline* type declaration of EPL, then FeatureJ ensures that if any variant of this *productline* must select *feature* ne if the *feature* ae is selected in that variant. The structural validation and constraint resolution is done on per product variant basis, rather than for an entire product line as suggested in [41].

***Validating bindings of FeatureJ types to meta-classes.*** The third step in type analysis in FeatureJ is to validate the binding of the *productline, variant*, and *feature* types and generate individual variants. During the instantiation of PL and PLVariant objects, FeatureJ checks for existence of corresponding *productline* and *variant* type in the repository. The internal data structures in the FeatureJ compiler store the complete structures of any *productline* and *variant* types defined in the program as well as feature containments (i.e., code fragments) along with the feature names. This repository essentially helps in reifying all feature related information so that valid program variants may be generated during the execution of a FeatureJ program.

Thus, given that Neg (Listing 1.5) is one of the SPL classes for which an abstract class Neg (1.8) was generated in Main\testEPL folder, then for the *variant* type negateExpression, an implementation is generated in the negateExpression\testEPL folder as shown in Listing 1.9.

**Listing 1.9: Implementation of Neg for the variant negateExpression**

```
1 package testEPL;
2 import com.unimag.sag.featurej.dynamic.*;
3 public class NegImpl extends Neg {
4   Exp expr;
5   public NegImpl(Exp e) {
6     super();
7     expr = e;
```

```
 8   }
 9   public void print() {
10     System.out.print("-(");
11     expr.print();
12     System.out.print(")");
13   }
14   public int eval() {
15     return expr.eval() * -1;
16   }
17 }
```

The addExpression *variant* type does not contain the *features* np and ne, consequently the implementation of the EPL class Neg is generated with default method implementations that raise errors as shown in Listing 1.10. This implementation resides in the addExpression\testEPL folder.

**Listing 1.10: Implementation of Neg for the variant addExpression**

```
 1 package testEPL;
 2 import com.unimag.sag.featurej.dynamic.*;
 3 public class NegImpl extends Neg {
 4   public void print() {
 5     throw new Error("This method has not been
 6     implemented for " + getClass().getName());
 7   }
 8   public int eval() {
 9     throw new Error("This method has not been
10     implemented for " + getClass().getName());
11   }
12 }
```

The implementations shown in Listings 1.9 and 1.10 are available in the PLVariant objects NExpr and AExpr. A Neg object of negateExpression *variant* is obtained via a PLVariant object (such as NExpr PLVariant object that encapsulates negateExpression variant) using the getVariantObject method as shown in Listing 1.7.

***Exact error reporting in FeatureJ.*** FeatureJ uses the Java semantics modules of JastAdd for semantically checking each of the newly generated classes of each variant. Any inconsistency in the composed code, such as a missing declaration when its access has been introduced by the selection of features, is found in the fourth step of FeatureJ type analysis. The error checking in FeatureJ is not restricted to finding access and declaration inconsistencies. JastAdd implements semantic error checking by calling the nameCheck(), typeCheck(), accessControl(), exceptionHandling(), checkUnreachableStmt(), definiteAssignment(), and checkModifiers() methods in that sequence on each AST node and its children starting with the Program node that represents the complete Java program including any class(es), interface(s), and packag(es) along with imported types in Java's default packages and any included jar files. We extend this mechanism to

bind error messages on specific AST nodes to the features that contain them. Any semantic error found in any of the above mentioned analyses, such as a wrong entity name introduced by a feature, wrong type for an entity in a statement contained in a feature, wrong access for any entity contained within a feature, as well any semantic errors related to exceptions, unreachable statements, unassigned variables, and modifiers related errors are found in type checking the variants. This analysis also takes into account the variant definitions in the FeatureJ program, so that FeatureJ is able to report variant specific error messages. This means that if a feature contains a semantic error but is not included in the definition of variant A, then error messages, if any, pertaining to the variant A will not contain this error. On the contrary, if a variant is defined to contain this feature then the error message is included in the error reporting for this variant. This analysis is invoked when a variant type in FeatureJ is bound to a PLVariant type, i.e, when composing a variant as well as when a variant is modified by adding or removing features.

#### Support for Static type analysis in FeatureJ

As shown in Figure 5, the default composition and modification of the FeatureJ types takes place at runtime. Consequently, the name and type analysis for the FeatureJ types is delayed until runtime in the default case. Using the command line option for the compile time error reporting, the type analysis of FeatureJ can be carried out statically at the compile time. This means that the entire sequence of 1) FeatureJ name resolution, 2) structural validation of variant types, 3) constraint resolution per variant type(s), 4) validation of the bindings of FeatureJ types to meta-classes, and 5) exact error reporting is carried out at compile time before executing the application main shown in Figure 5. If FeatureJ finds any errors, syntactic or semantic, it reports the errors and halts the process of compilation without executing the application main.

## 7 Discussion

In Sections 5 and 6, we have presented how features are represented and composed in FeatureJ. Here, we lay out the differences between other feature approaches and FeatureJ in terms of feature representation and feature composition and state the main advantages of FeatureJ.

### 7.1 Differences With Other Feature Approaches

FeatureJ differs from the other feature approaches in terms of nature of features and consequently the possibilities of features as a modularity mechanism of its own right as enlisted below:

***Preprocessors and annotations*** FeatureJ attempts to obtain the maximum possible abstraction for features and related concepts by elevating them to first-class status in Java. On the contrary, preprocessing approaches do not have any special representation for either of the conceptual and concrete features. The feature containments in FeatureJ are not simple annotations. This and other differences are enlisted in the following:

1. Unlike the preprocessors, FeatureJ contains a sophisticated representation of the feature abstraction. Features in FeatureJ are not merely pieces of code that are pre-processed for addition to a specific variant. Since preprocessors have no representation for features, variants, or product lines, it is extremely difficult to associate any type-checking for features thus implemented [28,39].
2. FeatureJ is not susceptible to syntax, type or behavioral errors like the pre-processor mechanisms because of its representation of features and their composition as an integral extension of the syntax and semantics of the host language Java.
3. Code laced with pre-processors directives gets obfuscated [28]. Similarly, multiple color annotations make it difficult to discern the code belonging to the same and different features. By extending the syntax of Java entities, FeatureJ makes the feature containments more readable than either of the pre-processors or annotations. The feature containments look like any block statements in Java.
4. Unlike pre-processors as well as annotations, features in FeatureJ are reusable. The features declared in a productline type are defined once in the code in terms of feature containments, and then reused in several variant definitions as required.

***Concern-specific modularization mechanisms*** FeatureJ contains different units of modularization represented in terms of productline, variant, and feature types. As opposed to using concern-specific modularization mechanisms to represent features, FeatureJ represents features natively in Java. In this respect, FeatureJ differs from these approaches as follows:

1. With FeatureJ, features themselves are the modularization mechanism used to implement features. This way only one additional level of abstraction needs to be addressed (cf. Section 3.2.2). Developers need a small learning curve to implement features using FeatureJ.
2. Features in FeatureJ do not loose legibility as no other concern-specific modularization mechanisms are used to implement features. The feature containments are minimally intrusive. Unlike the feature approaches using concern-specific modularization mechanisms such as teams [23], or units and atoms [31], feature related code fragments are not required to be specially packaged in various modular entities. This also makes FeatureJ better when evolving the product lines by adding new features or modifying the containments of previously existing features.
3. Since FeatureJ concentrates on providing only the feature related capabilities [29], the semantics of features in FeatureJ is not convoluted (cf. Section

3.3.2). Its expressive power can be extended as required to address advances in feature-oriented programming and feature models.
4. As opposed to concern-specific modularization mechanisms in which expressing the relationship between the conceptual and concrete features is a non-trivial task [16,31,33], FeatureJ provides an intuitive way to address this relationship. Since both the conceptual and concrete features are the same entities no extra work is required to represent the relationship between the two in FeatureJ.
5. Unlike the concern-specific modularization mechanisms in which once features are composed they can not be traced back to how they were composed [10,31] (such as the aspects or teams that were used to compose the features), FeatureJ retains the identity of features, enabling the developer to process features, variants, and productlines even after they have been composed.

***Structural forms of feature representation*** The main difference between the feature approaches using structural forms of feature representation and FeatureJ is that features are not represented external to the SPL classes. FeatureJ differs from structural feature approaches in the following ways:

1. Whereas features in approaches such as AHEAD and FeatureC++ are refinements [4,7], in FeatureJ they are modular entities of their own right.
2. FeatureJ does not arrange feature related code in folder hierarchies. The code fragments belonging to features are indicated in place by feature containments.
3. Whereas order of composition of refinements to classes is of vital importance to these approaches [2,7,41], in FeatureJ no ordering of features is required since code fragments belonging to features are contained in place. A developer needs to be aware of the specifics of code fragments, especially when adding refinements to methods, to specify correct ordering in the equation file. On the contrary, a variant type definition in FeatureJ does not require any assumption about the ordering of selected features [41].
4. As opposed to an equation file in AHEAD, which represents composition specification in terms of textual entries without any semantics associated with them, FeatureJ contains a native and type-checkable representation of the conceptual features.
5. Product lines, variants, and features are not addressable in these approaches, as they are in FeatureJ in terms of programmatic entities. In FeatureJ the representation of conceptual and concrete features is semantically strong enough to enable processing multiple product lines and variants at both compile time and runtime.

***Language representation for conceptual features*** FeatureJ differs from FDL and VML by the fact that both the conceptual and concrete features are represented at language level. In the following we enlist the differences with these approaches:

1. In FeatureJ both the conceptual and concrete features are integrated in the host language Java with a common semantics.

2. Unlike these approaches, features are not composed using code generators via UML mappings [16,33]. The composition process is part of the compiler that also contains the common syntax for productlines, variants, and features. This enables a more coherent composition mechanism that is easy to extend and does not depend on either additional UML representations or code generator specific idiosyncrasies.
3. Whereas FDL represents an external DSL, the representation of features in FeatureJ is implemented as a Java extension.This enables FeatureJ to also compose features a part of the compilation process instead of making use of code generators.

### 7.2 Advantages of FeatureJ

Our goal in creating FeatureJ was to realize a holistic feature approach in which the representation and the composition mechanism of feature related concepts is strong enough to support various possibilities related to features. A first-class representation in FeatureJ essentially set ups the background upon which various capabilities of features are effectively realized. In the following, we enlist the most important advantages of FeatureJ:

1. **Type support for features** - FeatureJ provides common representation and stream-lined composition for the conceptual and concrete features in terms of types. This enables FeatureJ to provide 1) implicit type-checking of variants and 2) the ability to address and manipulate multiple product lines and multiple variants in a single program.
2. **Error reporting and traceability** - Features in FeatureJ retain their identity even after composition. This enables FeatureJ to support exact error reporting not only at the compile time (static type checking) but also at the runtime (to report errors in a variant that is composed or modified at runtime).
3. **Extensibility** - FeatureJ compiler is based upon JastAdd, an extensible compiler architecture. FeatureJ's design is well structured so that its syntax can be easily extended to support the granularity of features to coarser or finer levels as required. Similarly, FeatureJ's semantics can be extended to keep up with advances in feature modeling concepts such as e.g., feature attributes and cardinalities.

As described in Sections 3.2, 3.3, and 7.1, FeatureJ reduces the complexity of implementing an SPL as compared to other feature approaches at the same time providing an intuitive way to program product lines, variants, and features. The feature containments in FeatureJ provide readability and maintainability for relating code fragments in SPL classes to specific features. The ease with which multiple product lines and multiple variant can be created, modified, and queried increases the usability of FeatureJ.

## 8 Conclusion and Further Work

We have applied FeatureJ to various product lines such as the GPL, the EPL, and the NPL[12]. These and other product lines are used as testbeds for testing various functionalities provided by FeatureJ. FeatureJ completely removes the need for mapping the conceptual features to the concrete features, because both are represented by the same entities, i.e. FeatureJ types. Furthermore,the feature composition is part of FeatureJ compilation (including name and type analysis). Thus, we achieve our goal to create a uniform representation and a coherent composition of features.

### Extending FeatureJ

FeatureJ is highly extensible due to its implementation as an extension of JastAdd. Although FeatureJ can be extended in a variety of ways, we discuss only two extensions, one to the representation of features in FeatureJ and the other to the composition of features in FeatureJ.

***Extending the representation of features*** Since FeatureJ integrates the conceptual and concrete features into a uniform representation, advances in feature modeling can be implemented as extensions to the FeatureJ syntax and semantics. Czarnecki et al. [12,13] suggested a number of extensions to the original feature modeling concepts, such as staged configuration, cardinality based features (where mandatory and optional features are special cases), feature groups (alternative features, inclusive-or features etc.), attributes (type association of features), modularization, etc. With FeatureJ this translate into extensions to the syntax of FeatureJ types (such as feature cardinality and feature groups) and semantics of the FeatureJ types (such as attributes - FeatureJ types provide a unique opportunity for FeatureJ to Java type association). Modularization of feature models mean models with leaves that are themselves other feature models. This suggests composition of multiple *productline* types. We intend to explore such feature modeling extensions to FeatureJ.

***Extending the composition of features*** One of the most important avenues of further development in FeatureJ is to carry out validation of all the variants of a given product line. As discussed in Section 6.3, currently FeatureJ supports semantically checking an individual variant that is defined as a *variant* type, and bound to a PLVariant object. Many researchers have suggested approaches for the safe composition of an entire product line [15,27,41]. These approaches are either formal, or supported through separate tools. A natural extension of the type semantics in FeatureJ would be to integrate safe composition of all the possible variants of a *productline* type. Gheyi et al. [19] have suggested an approach in which feature models are represented in the alloy language. The alloy analyzer is used to 1) checking a configuration (variant) against a feature

[12] http://firstclassfeatures.org/index.php?n=Examples.FeatureJ

model (similar to functionality currently provided by FeatureJ) and 2) find all configurations (i.e., checking all variants of a product line). The unified representation of features in FeatureJ provide a unique opportunity to integrate alloy transformations of FeatureJ types along with reified information available in FeatureJ meta-classes. The information about entity declaration and entity accesses/references is central to JastAdd's analyses and also the basis of name resolution and type analysis in FeatureJ. FeatureJ already provides information about access and declarations of various entities in a particular *feature*[13]. Apart from the feature relations, implicit, and explicit constraints specified in [19], the dependencies between features (e.g., feature A requires feature B because feature A contains a method call of a method defined in feature B, called verification properties in [41] which are essentially the required declarations for given accesses/references) can be included as additional constraints in alloy model checking. The FeatureJ type declarations can be easily transformed to alloy declarative models (e.g., alloy models of features as discussed in [19]). We intend to implement and test safe composition capabilities of FeatureJ against different product lines.

**ACKNOWLEDGMENT**

We thank Sven Apel and Christian Kästner for their valuable comments on an earlier draft of this paper. Sagar Sunkle is a PhD candidate at the Database group of Otto-von-Guericke University of Magdeburg and receives scholarship from the federal state of Saxony-Anhalt in Germany. Sebastian Günther works with the Very Large Business Applications lab of Otto-von-Guericke University of Magdeburg. The Very Large Business Applications Lab is supported by SAP AG.

## References

1. S. Apel and C. Kästner. An overview of feature-oriented software development. *Journal of Object Technology (JOT)*, 8(5):49–84, July/August 2009. Guest Column.
2. S. Apel, C. Kästner, A. Größlinger, and C. Lengauer. Type-safe feature-oriented product lines. Technical Report MIP-0909, Department of Informatics and Mathematics, University of Passau, 2009.
3. S. Apel, C. Kästner, and C. Lengauer. FEATUREHOUSE: Language-independent, automated software composition. In *Proceedings of the 2009 IEEE 31st International Conference on Software Engineering-Volume 00*, pages 221–231. IEEE Computer Society Washington, DC, USA, 2009.
4. S. Apel, T. Leich, M. Rosenmüller, and G. Saake. FeatureC++: On the Symbiosis of Feature-Oriented and Aspect-Oriented Programming. In *Proceedings of the International Conference on Generative Programming and Component Engineering*, pages 125–140. Springer, 2005.

[13] See the graph product line example - http://firstclassfeatures.org/index.php?n=FeatureJ.GraphPL

5. S. Apel, T. Leich, and G. Saake. Aspectual feature modules. *IEEE Transactions on Software Engineering*, 34(2):162–180, 2008.
6. S. Apel and C. Lengauer. Superimposition: A Language-Independent Approach to Software Composition. In *Software Composition: 7th International Symposium, SC 2008, Budapest, Hungary, March 29-30, 2008. Proceedings*, pages 4–19. Springer-Verlag New York Inc, 2008.
7. D. Batory. Feature-Oriented Programming and the AHEAD Tool Suite. In *Proceedings of the 26 th International Conference on Software Engineering*. ACM, 2004.
8. D. Batory. Feature Models, Grammars, and Propositional Formulas. In *Proceedings of the International Software Product Line Conference*, volume 3714 of *Lecture Notes in Computer Science*, pages 7–20. Springer, 2005.
9. D. Batory, J. Liu, and J. Sarvela. Refinements and multi-dimensional separation of concerns. *ACM SIGSOFT Software Engineering Notes*, 28(5):48–57, 2003.
10. R. Chitchyan, I. Sommerville, and A. Rashid. A Model for Dynamic Hyperspaces. In *Workshop on Software engineering Properties of Languages for Aspect Technologies: SPLAT (held with AOSD*, 2003.
11. K. Czarnecki and U. Eisenecker. *Generative Programming: Methods, Tools, and Applications*. Addison-Wesley, 2000.
12. K. Czarnecki, S. Helsen, and U. W. Eisenecker. Formalizing cardinality-based feature models and their specialization. *Software Process: Improvement and Practice*, 10(1):7–29, 2005.
13. K. Czarnecki, S. Helsen, and U. W. Eisenecker. Staged configuration through specialization and multilevel configuration of feature models. *Software Process: Improvement and Practice*, 10(2):143–169, 2005.
14. M. de Jonge and J. Visser. Grammars as feature diagrams. Apr. 2002. draft.
15. B. Delaware, W. Cook, and D. Batory. A machine-checked model of safe composition. In *Proceedings of the 2009 workshop on Foundations of aspect-oriented languages*, pages 31–35. ACM New York, NY, USA, 2009.
16. A. v. Deursen and P. Klint. Domain-specific language design requires feature descriptions. *Journal of Computing and Information Technology*, 10(1):1–17, 2002.
17. S. Ducasse, O. Nierstrasz, N. Schärli, R. Wuyts, and A. Black. Traits: A Mechanism for Fine-Grained Reuse. *ACM Transactions on Programming Languages and Systems*, 28(2):331–388, 2006.
18. T. Ekman and G. Hedin. The JastAdd system - modular extensible compiler construction. *Science of Computer Programming*, 69(1-3):14–26, 2007.
19. R. Gheyi, T. Massoni, and P. Borba. A theory for feature models in alloy. In *First Alloy Workshop*, pages 71–80, 2006.
20. S. Günther and S. Sunkle. Enabling feature-oriented programming in ruby. Technical Report FIN-017-2009, Very Large Business Application Lab, University of Magdeburg, Germany, Nov. 2009.
21. S. Günther and S. Sunkle. Feature-oriented programming with ruby. In *Proceedings of the First Workshop on Feature-Oriented Software Development (FOSD)*. ACM Press, OCT 2009.
22. G. Hedin and E. Magnusson. Jastadd: an aspect-oriented compiler construction system. *Sci. Comput. Program.*, 47(1):37–58, 2003.
23. C. Hundt, K. Mehner, C. Pfeiffer, and D. Sokenou. Improving Alignment of Crosscutting Features with Code in Product Line Engineering. *Journal of Object Technology (JOT)–Special Issue: TOOLS EUROPE*, 6(9):417–436, 2007.

24. C. Kaewkasi and J. Gurd. Groovy AOP: a dynamic AOP system for a JVM-based language. In *Proceedings of the 2008 AOSD workshop on Software engineering properties of languages and aspect technologies*, page 3. ACM, 2008.
25. K. Kang, S. Cohen, J. Hess, W. Novak, and A. Peterson. Feature-Oriented Domain Analysis (FODA) Feasibility Study. Technical Report CMU/SEI-90-TR-21, Software Engineering Institute, Carnegie Mellon University, 1990.
26. C. Kästner and S. Apel. Integrating compositional and annotative approaches for product line engineering. In *Proceedings of the GPCE Workshop on Modularization, Composition and Generative Techniques for Product Line Engineering (McGPLE)*, number MIP-0802, pages 35–40. Department of Informatics and Mathematics, University of Passau, Oct. 2008.
27. C. Kästner and S. Apel. Type-checking software product lines - a formal approach. In *Proceedings of the International Conference on Automated Software Engineering*. IEEE Computer Society, 2008.
28. C. Kästner and S. Apel. Virtual separation of concerns – a second chance for preprocessors. *Journal of Object Technology (JOT)*, 8(6):59–78, Sept. 2009.
29. C. Kästner, S. Apel, and D. Batory. A Case Study Implementing Features Using AspectJ. In *Proceedings of the International Software Product Line Conference*, pages 223–232. IEEE Computer Society, 2007.
30. C. H. P. Kim and K. Czarnecki. Synchronizing Cardinality-Based Feature Models and Their Specializations. In *European Conference on Model Driven Architecture Foundations and Applications*, pages 331–348. Springer, 2005.
31. R. Lopez-Herrejon, D. Batory, and W. R. Cook. Evaluating Support for Features in Advanced Modularization Technologies. In *Proceedings of the European Conference on Object-Oriented Programming*, volume 3586 of *Lecture Notes in Computer Science*, pages 169–194. Springer, 2005.
32. R. E. Lopez-Herrejon. The expression problem as product-line and its implementation in AHEAD. Technical report, Department of Computer Sciences, University of Texas at Austin, October 2004.
33. N. Loughran, P. Sánchez, A. Garcia, and L. Fuentes. Language support for managing variability in architectural models. In *Software Composition*, pages 36–51, 2008.
34. S. McDirmid, M. Flatt, and W. C. Hsieh. Jiazzi: New-Age Components for Old-Fashioned Java. In *Proceedings of the International Conference on Object-Oriented Programming, Systems, Languages, and Applications*, pages 211–222. ACM Press, 2001.
35. M. Mezini and K. Ostermann. Variability Management with Feature-Oriented Programming and Aspects. In *Proceedings of the International Symposium on Foundations of Software Engineering*, pages 127–136. ACM Press, 2004.
36. H. Ossher and P. L. Tarr. Hyper/j: multi-dimensional separation of concerns for java. In *ICSE*, pages 734–737, 2000.
37. C. Prehofer. Feature-Oriented Programming: A Fresh Look at Objects. In *Proceedings of the European Conference on Object-Oriented Programming*, volume 1241 of *Lecture Notes in Computer Science*, pages 419–443. Springer, 1997.
38. M. Rosenmüller, N. Siegmund, G. Saake, and S. Apel. Code Generation to Support Static and Dynamic Composition of Software Product Lines. In *Proceedings of the 7th International Conference on Generative Programming and Component Engineering*. ACM Press, Oct. 2008.
39. O. Spinczyk and D. Beuche. Modeling and building software product lines with eclipse. In *OOPSLA '04: Companion to the 19th annual ACM SIGPLAN confer-*

*ence on Object-oriented programming systems, languages, and applications*, pages 18–19, New York, NY, USA, 2004. ACM.

40. S. Sunkle, M. Rosenmüller, N. Siegmund, S. S. ur Rahman, and G. Saake. Features as First-class Entities – Toward a Better Representation of Features. In *GPCE Workshop on Modularization, Composition and Generative Techniques for Product Line Engineering (McGPLE)*, pages 27–34. Department of Informatics and Mathematics, University of Passau, Oct. 2008.
41. S. Thaker, D. Batory, D. Kitchin, and W. Cook. Safe composition of product lines. In *Proceedings of the 6th international conference on Generative programming and component engineering*, pages 95–104. ACM, 2007.