# Features and Feature Models: A Survey of Variation Representations

Eric Walkingshaw
School of EECS
Oregon State University
walkiner@eecs.oregonstate.edu

## Abstract

*This survey explores and organizes existing work on the long-term management of software that varies in multiple dimensions. It focuses, in particular, on the representation of features in software product lines, and on capturing relationships between features in feature models.*

## 1 Introduction

Managing variation in code and other artifacts is a fundamental problem in software engineering that manifests itself in many different ways throughout the field. This leads to many branches of research and a broad assortment of tools for managing different types of variation, each with their own ways of representing the parts of a system that vary and how those parts can be combined.

While the diversity of variation representations is not inherently a problem, it would be nice if advances in one of line of research could be easily understood and incorporated by others. Additionally, by focusing on domain-specific applications, researchers may miss insights that can only be gained by examining the problem from a broader, more abstract point of view.

Toward a solution to these issues, we have been developing a general model for representing software variation, the *choice calculus* [11]. Our goals for the choice calculus are to provide a common language of discourse and a theoretical foundation for tools and research on software variation management, filling a role similar to the lambda calculus for programming languages or relational database theory for databases.

While our approach in developing the choice calculus in fundamentally and purposefully top-down, a knowledge of the research we hope to support is crucial to our success. This paper will attempt to organize and explicate one multifaceted branch of software variation research, specifically, the development and maintenance of *long-term*, *multi-dimensional* variations.

### 1.1 Scope of Work

The scope of this survey is best understood by comparison with what is omitted. A *short-term* variation might represent a temporary exploration of alternatives by a designer, perhaps to circumvent a premature commitment. For example, a designer may not know which data structure to use in a particular instance, and so provides two alternatives, one of which will eventually be chosen when the operations required of the data structure are more clear.[1] In contrast, a long-term variation is typically planned, intended to produce distinct programs, and must be maintained. For example, multi-platform software might contain parts that correspond to only the Windows or only the Unix versions of the program.

While this work examines multi-dimensional variation tools, revision control and software config-

[1]Short-term variations are currently very poorly supported by tools, despite evidence that they are needed [31]. Identifying and addressing gaps in support is another potential benefit of the top-down approach.

uration management (SCM) systems instead focus on variation management in a *single dimension*, allowing users to easily track and manage variation over *time* [12]. These variations are usually captured as line-based *deltas*, or *patches*, which indicate the minimum number of lines that differ between two versions of a file [28]. Many SCM systems actually do provide support for additional dimensions of variation (e.g. *release number* or *target platform*) via repository branching operations, but this is neither their primary focus nor their forte [7], as evidenced by the organizational rigor needed to successfully manage multiple repository branches [33]. Multi-dimensional approaches contrast this by treating all dimensions of variation equally.

### 1.2 Software Product Lines

Long-term, multi-dimensional variation management is closely related to the idea of *software product lines* (SPLs) (also called *product families*) [27]. Very generally, a software product line is a collection of related programs based on a common software platform or generated from a common set of resources. The concept of product lines originated in manufacturing to minimize the development and production costs of a range of similar products—the classic example is car manufacturers who design many different car models around a common set of components (e.g. a small set of different chassis and engines), and allow further customization for individual customers through feature options (e.g. transmission type, air conditioning, power windows). The first to apply this idea to software engineering appears to have been Parnas [26]. Since then, it has become by far the dominant paradigm in variation management.

Essentially all current research in tool-supported multi-dimensional variation management is directed toward the creation and maintenance of SPLs, but this research is only a small part of SPL research on the whole. Much research focuses on the methodology and organizational aspects of SPL engineering [9, 24, 27]. While this work can inform the development of tools which support these processes, it is of little interest in a survey of variation representations. It is also worth noting that most work seems to assume that SPL-oriented variation management and SCM are fundamentally different problems, while we have assumed in developing the choice calculus that they are two views of the same problem. The relationship of SCM to SPL development is made explicit by Krueger in [22].

Because a SPL is such a high-level idea, we can mostly ignore it while discussing lower-level variation representations. Much of the discussion, however, will revolve around a few key SPL concepts. Most importantly, a *feature* is some piece of functionality which can be included or not in a program, and a *feature model* is a way of describing the relationships and constraints that exist between features. Together these form two mostly independent levels of variation representation in a SPL. The lower-level feature representation captures the actual code (or other data) that constitutes a feature, and how it is added to the program to incorporate the feature. The higher-level feature model describes relationships between features, such as alternative features which form a dimension, and describes which lists of features can be selected to produce valid programs. Theoretically, one could choose a feature representation and feature model separately and combine them to form a complete variation representation. Finally, a *base program* is just a program containing no features, which features can then be added to.

In the rest of this paper we will examine several different ways of representing both features and feature models. In Sections 2, 3, and 4 we will look at compositional feature representations, where features are defined and exist separately from the base program. In Section 5 we will see annotative approaches, where features are embedded within the context of the base program. In Section 6 we look at representations of feature models, and wrap up with some conclusions in Section 7.

## 2 Compositional Feature Representations

Compositional approaches to feature representation are related to many traditional ideals of *software reuse*. In particular, compositional methods emphasize the *modularization* of features—that the code and other data comprising a feature should be self-contained and separate from code describing other features and the base program. This leads to a highly distributed variation model where the variational structure is represented by one or more base programs, a set of discrete features, and some kind of feature model for describing constraints on which features can be applied to the base program and in what order.

In general, compositional approaches sacrifice variational *granularity*, *flexibility*, and *generality* for higher degrees of *maintainability* and *comprehensibility*. As we will see in the next two sections, compositional approaches typically cannot represent arbitrarily fine-grained variation, nor can they represent variation in arbitrary places in the program. Additionally, working tools are tied to specific programming languages, or require an extension for working with each new type of artifact. Annotative approaches do not have these limitations, but they are accepted in compositional approaches because it is thought that modularizing features makes them easier to maintain and understand,[2] due to a separation of concerns [32].

The goal of modularization towards increased maintainability and comprehensibility is closely related to the idea of *stepwise refinement* [34], in which a program is incrementally built by applying refinements that correspond to distinct development phases or design decisions. Some of the tools described in this section primarily aim to support stepwise refinement and have merely been adapted for use in SPL development, while others have been designed with both uses in mind.

In the next two sections we will examine two different ways of expressing and composing features. The first relies on concepts of inheritance and mixins from object-oriented programming, while the second uses strategies from aspect-oriented programming. While "features" are a general concept that apply in any SPL system, the emphasis on expressing and isolating features has led to the term *feature-oriented programming* (FOP) to describe compositional approaches to variation management. Researchers in aspect-oriented methods constrain this term even further, using it to refer exclusively to inheritance-based, object-oriented feature representations. Since this is often convenient, we will adopt this usage as well.

## 3 Feature-Oriented Programming

The goals of compositional feature representations, to separate and encapsulate features, are very similar to the goals of object-oriented programming (OOP), so it should perhaps not be surprising to see object-oriented techniques prevalent in the compositional approach. In fact, the notion of *inheritance*, fundamental to OOP, can be viewed as a limited variation mechanism in itself. Inheritance allows a subclass to add data and functionality to a base class, and to change the existing behavior of a base class through method overriding. Subclasses can vary the same base class in different ways, and clients can then choose which subclass (variant) to instantiate and use.

There are many limitations of a purely inheritance-based variation model. First, a feature may span multiple subclasses, making it difficult to ensure that all of the right subclasses are instantiated to incorporate a particular feature—this limitation is easily handled by the tools described below. More fundamental limitations also exist, however, such as the inability to choose a subset of classes in a particular class lineage in order to include some features and exclude others. These problems can be partially solved with design patterns (e.g. the decorator pattern, for the above example) [13], but this requires foresight and cannot be applied to existing, unmodifiable code. As a more general solution, traditional inheritance is often supplemented in FOP with *mixins* [8]. A mixin is essentially a subclass with no explicit superclass. When applied to an existing class, a mixin's functionality is transparently incorporated into the original class.

### 3.1 GenVoca

The GenVoca feature-oriented design methodology [5] relies heavily on subclasses and mixins,

[2]Though removing code from context can also decrease understandability. See Section 5.

referring to them collectively as "refinements". In GenVoca, a base program is a set of classes, while a feature is a set containing new classes to add to the program and refinements to apply to existing classes. GenVoca calls base programs and features, "constants" and "functions", respectively.

Constraints on the ordering of feature application to a base program are very common in this approach, and GenVoca provides a simple, declarative feature algebra for specifying these constraints and defining valid combinations of features; we will come back to this algebra in Section 6.2.

The code-level details of applying a feature to a base program (e.g. determining which refinements to apply to which classes) are left undefined, ostensibly to be flexible with regard to the object language. This exemplifies a frequent trade-off in generality that FOP systems make: a fully specified representation ties a model to a particular type of artifact, while bids for generality leave gaps in the model. GenVoca's strategy of defining everything but one artifact-specific operation is a typical solution, reminiscent of the framework model.

## 3.2 Multi-Dimensional Separation of Concerns

An implementation of GenVoca for Java programs has been built with Hyper/J [25], which in turn is a Java realization of Tarr et al.'s *Multi-Dimensional Separation of Concerns* (MSC) [32]. MSC has been hugely influential for explicating the problems that FOP is meant to address and establishing a terminology and set of goals for FOP researchers.

The variation model developed in MSC is very general and much further removed from any particular object language than GenVoca. MSC advocates an intentionally coarse variation granularity and the use of standard modules (as defined by the object language, e.g. packages and classes in Java), without additional language extensions. Modules can be grouped together into a "hyperslice", which isolates a single concern or feature. A hyperslice is so named because it "slices" across potentially many modules in the program; that is, a single module in the composite program can be expected to exist in many hyperslices. Sets of hyperslices can be grouped into dimensions. And finally, an object language-specific composition operation must be provided to compose hyperslices.

As an example of how this model actually works, in Java a hyperslice might be realized by a directory tree full of class files. Many of the class names in each hyperslice will be duplicated. If a class named *C* in one hyperslice contains method *a*, and a corresponding class also named *C* in another hyperslice contains method *b*, when we compose the two hyperslices, we would expect the resulting class *C* to contain both methods *a* and *b*.

## 3.3 AHEAD

The last inheritance-based system we will look at is AHEAD [6], which is both the direct successor of GenVoca and a realization of many of the goals established in MSC. AHEAD represents the state-of-the-art in FOP systems. The primary advances in AHEAD over GenVoca are increased artifact structure leading to clearer composition semantics and better abstraction, and a generalization of refinement to other, potentially non-code artifacts.

In GenVoca, a constant corresponded to a set of classes and a function to a set of classes and refinements. In AHEAD these components have more structure. A constant is a base artifact (e.g. a class in an OOP language) or a list of constants; a function is a refinement or a list of constants and/or refinements. In other words, constants and functions are trees. Applying a function to a constant is then just a tree merge operation, where corresponding elements are composed. Composition of internal nodes, which are represented by directories in the implementation, is implemented by composing corresponding children and adding the union of all remaining children as children of the new composite node. When composing two leaves, one should be a refinement and the other a constant; the refinement is then simply applied to the constant.

AHEAD also generalizes refinement to new artifact types, but this is not very ground-breaking since it still requires a plug-in to implement mixin-like functionality for every artifact type it intends to support. Still, simply supporting multiple artifact types in an inheritance-based feature representation appears to be unique.

## 4 Aspect-Oriented Programming

The primary goals of the compositional approach to feature representation are to *separate* and *localize* features from the rest of the program, but FOP approaches often struggle with both. Separating a feature from the surrounding program is sometimes difficult because of the coarseness of variation mechanisms like class refinement with method overriding. Features often overlap with other features and don't correspond exactly to a set of classes or methods.

Isolating features into sets of classes and methods may be possible with refactoring, but we risk obscuring the natural structure of the code. Alternatively, we can simply duplicate common parts in many feature representations. Duplication seems to be the more common approach, but it is inefficient and significantly impairs maintainability. Changes in a duplicated part of one feature must be propagated to other features, increasing the amount of work that must be done and increasing the risk of errors if one duplicate is edited and another is not.

Even if we do successfully separate a feature with FOP, it may still be poorly localized. A feature may be spread across many modules, with only minor changes in each one. A classic example is adding logging to an application, which may affect nearly every method but require adding only a single line to each. In a model like AHEAD, this feature would lead to the structure of the *entire* source tree being duplicated, with each class refinement looking something like the examples in Figure 1. Notice that, every time we add a class or method to the underlying system, we must also add that class or method to the logging feature. It is also impossible to apply the logging feature to a different program, since the feature contains an overwhelming amount of base program-specific information.

Aspect-oriented programming (AOP) [20] is dedicated to concisely representing exactly these kinds of so-called *crosscutting* features. An *aspect* is in some ways similar to an object in that it can have its own state and methods, but aspects are not directly instantiated or manipulated. Instead, they rely on hooking into the control-flow of the underlying (object-oriented) program, usually by intercepting method calls. This will become clearer in the following subsections as we examine two Java language extensions that support AOP.

```
class Logger {
  static void log(String s) {
    System.out.println(s);
  }
}
refine class Cat {
  void purr() {
    Logger.log("Cat.purr()");
    super.purr();
  }
  Affection pet() {
    Logger.log("Cat.pet()");
    return super.pet();
  }
  ...
}
refine class Dog {
  HospitalBill bite(Child c) {
    Logger.log("Dog.bite()");
    return super.bite(c);
  }
  ...
}
...
```

**Figure 1:** Adding logging with FOP.

### 4.1 AspectJ

AspectJ [19] is almost certainly the most widely used AOP system and can be regarded as the defacto standard for AOP implementations. In AspectJ, an aspect is a stand-alone module, intended to encapsulate a single, crosscutting feature. Like an object, an aspect can have data members and methods—these are essentially local variables and procedures, respectively, since aspects cannot be instantiated. Additionally, an aspect can contain *pointcut* definitions and *advice*. A pointcut describes a set of *join points*, which are places in the control flow where new code can be inserted and executed. Advice specifies code to run at the join points described by a particular pointcut.

```
aspect Logger {
  void log(String s) {
    System.out.println(s);
  }
  pointcut all() : execution(* *.*(..))
  before(String c, String m): all()
      && methodClass(c)
      && methodName(m) {
    log(c ++ "." ++ m ++ "()");
  }
}
```

**Figure 2:** Adding logging with AOP.

A join point is like a hook into the control flow of the underlying object-oriented program. They are points in the execution of a program where an aspect can insert code (advice). Examples of join points include method execution, class instantiation, and member variable referencing.

A pointcut is essentially a predicate over join points, and the set of join points it describes is the set of all join points that fulfill the predicate. A pointcut can be either a primitive pointcut or a boolean expression of other pointcuts. There are many primitive pointcuts: some match on the types involved at a particular join point (e.g. the receiver, argument, or return types of a method call), while others match the names of elements at a join point (e.g. the name of an executed method or accessed data field). Defining pointcuts in this way is both very powerful and potentially dangerous. Relying on the names of methods and data fields forces programmers to adhere to a set of non-enforceable conventions or risk breaking pointcut definitions. Newer versions of AspectJ can instead match on Java 5 annotations, which are more explicit and stable than pointcuts. Stable join points provide a considerably safer and more structured way to insert advice [18].

Advice can be executed before, after, or around each join point captured by a particular pointcut—for example, if the pointcut *n* intercepts calls to the `Nukes.launch()` method, advice may specify that a security check be performed on the caller before *n* is executed, and invoke the `duckAndCover()` method on every `Person` object after.

Figure 2 demonstrates an aspect-oriented representation of the logging feature discussed above. The statement beginning with the keyword `pointcut` introduces a new pointcut named `all` that matches all method invocations—`execution` is a primitive pointcut operator provided by AspectJ that takes a pattern and uses it to match method signatures. The block beginning with the `before` keyword is advice to be executed before the pointcut provided to the right of the colon. The `all` pointcut matches all method invocations and the `targetClass` and `methodName` pointcuts (defined elsewhere) match all join points, and bind their argument pointers to the class name and method name of the invoked method, respectively.

When applied to a base program, the AOP feature in Figure 2 and the FOP feature in Figure 1 produce equivalent programs, but the AOP version encapsulates the feature much better. It is much easier to maintain, since changes to the base program do not require changes to the feature representation, and it can even be applied to other base programs.

AspectJ is not designed with SPLs or variation management in mind, but rather to support modularization and stepwise refinement. In particular, aspects are simply included in a system; there is no notion of aspect selection or relationships between aspects (other than application ordering dependencies). This is not a fundamental limitation of aspects, however. As our little example suggests and as multiple researchers have observed [2, 23], AOP and FOP are very complimentary—often the weaknesses of one approach correspond to strengths of the other. Thus, it would be very beneficial for variation management systems to incorporate both. Apel et al. have begun work on this by adding aspects to traditional feature models in [2]. Next we will see a somewhat more radical approach, where mixin and aspect language constructs are merged.

## 4.2 Caesar

The goal of the Caesar programming language [23], an extension to Java, is to combine and improve on the strengths of FOP and AOP to provide language-level support for variability, feature and refinement modularity, and SPL development.

Caesar classes, called *crosscutting layers*, are generalized to include aspect-oriented features (pointcuts and advice) as well as several other experimental extensions to the class metaphor. With both FOP and AOP features available, users could simply pick the metaphor that works better for a particular feature, but the designers claim that it offers significantly more than just the sum of these parts. As an example, recall that in Section 4.1 we noted that aspect member variables and methods correspond to variables and procedures in a procedural language—that is, in Java parlance, they are implicitly `static`. Thus, if one wants to refactor an instance variable out of a class and into an aspect, the aspect must manually maintain a mapping from object instances to values of the original instance variable type. This is cumbersome, error-prone and not very object-oriented. The alternative is to leave the instance variable in the original class, but if it is specific to the feature described by the aspect, we have violated our goal of separating concerns. Since Caesar combines aspects and mixins, a layer can both describe aspect-oriented functionality while also mixing new state and methods into existing classes, encapsulating the feature more cleanly than either FOP or AOP could alone.

Caesar expresses variability through a notion of *deployment*. A layer can be initially deployed, or can be deployed dynamically at run time simply by instantiating the layer and applying a special `deploy` operation to it. When layers correspond to features, deployment corresponds to feature selection. Since layer deployment is managed within the Caesar language itself, arbitrary relationships can be described between features. For example, one could represent a dimension as a list of layer instances; one instance is selected from the list and then deployed. Caesar's deployment scheme is interesting for a few reasons: First, it provides a way to selectively apply aspects, something which is not supported by AOP tools. Second, it appears to be unique among feature-oriented tools in supporting and promoting dynamic feature selection.

Caesar provides many other esoteric language features, such as "bidirectional interfaces" and "virtual classes", mostly directed toward isolating features more completely from the base program. These can lead to improvements in feature reuse, but are excluded here to retain some sense of brevity.

While Caesar is perhaps the most complex variation representation we have seen yet, there are still things it cannot do. In the next section we reign in the complexity, lose a little bit of modularity, and present a class of simple but highly expressive feature representations.

## 5 Annotative Feature Representations

Annotative approaches represent features by marking corresponding sections of the source code (or other data) in some way. Unlike compositional representations, annotations are often completely independent of the object language, and can therefore capture variation in a much broader range of artifacts. They are also usually capable of very fine-grained variation, and of capturing variation distributed arbitrarily throughout the artifact or even across artifact types. In other words, the representational design goals of annotative approaches are almost completely opposite of those for compositional representations. In general, annotative representations exalt granularity, flexibility and generality, and are willing to sacrifice modularity and maintainability to maximize them.

Comparing the comprehensibility of both types of representation is a bit more subtle. In general, as the complexity of a variational structure increases, we expect the degradation of comprehensibility to be slower with compositional representations. Thanks to a separation of concerns, compositional approaches make it easier to understand complex structures by simply ignoring the parts we aren't interested in. However, for simpler structures, with fewer dimensions of variation and less overlapping concerns, we might expect annotative approaches to be easier to understand, if the annotations themselves are not very intrusive. While compositional approaches require switching between features and the base program to understand how they will interact, annotative approaches allow users to view features in context, with other features and the base program.

Kim et al. argue that annotative approaches are preferable when *refactoring* an existing program to extract a set of features [21]—they show that the number of features, and particularly the number of nested features, is relatively small for programs designed without features and SPLs in mind. However, they also concede that projects designed with variation in mind from the start are likely to be considerably more complex.

The following tools are just a few of many annotative approaches. Each is representative of a potentially larger class of related tools. The first is a purely annotation-based tool, the second relies on both user-interface and language-specific support, and the last attempts to combine annotations with feature composition to get the benefits of each.

## 5.1 C Preprocessor

One of the most straightforward, and certainly the most widely used [10], annotation-based variation tool is the conditional compilation subset of the C Preprocessor language [14].

Despite the name, CPP is almost completely indifferent to the type of underlying artifact—it must simply be a text file, and not contain text that resembles CPP syntax. Although CPP can do other things, like macro definition and expansion, we are interested only in its ability to conditionally include parts of files. CPP provides a set of directives for this purpose, which combine to form conditional statements in the obvious way: `#if`, `#ifdef`, `#ifndef`, `#else`, `#elif`, and `#endif`. Each directive must occur on a line by itself, and text between directives form the then- and else-blocks of the conditional structures. The `#if` and `#elif` directives can accept arbitrary C-style integer expressions as their conditions—if an expression evaluates to a non-zero value, the corresponding block of text is included; if it evaluates to zero the block is not included and either the next `#elif` is checked, a corresponding else-block is included, or nothing is included, depending on the conditional structure. The `#ifdef` and `#ifndef` directives accept a *macro* as an argument. This macro can either be "defined" or "undefined". A defined macro causes the block of an `#ifdef` to be included, while an undefined macro causes the block of an `#ifndef` to be included. When the preprocessor is run on a given file, the user specifies a set of macros which will be considered defined.

```
class Buffer {
  int buf;
#ifdef RESTORE
  int back;
#endif
  void set(int b) {
#ifdef LOG
    Logger.log(buf);
#endif
#ifdef RESTORE
    back = buf;
#endif
    buf = b;
  }
#ifdef RESTORE
  void restore() {
#ifdef LOG
    Logger.log(buf);
#endif
    buf = back;
  }
#endif
}
#ifdef LOG
class Logger {
  static void log(int i) {...}
}
#endif
```

**Figure 3:** Buffer with optional logging and restore features in CPP.

The limitation that directives must occur on a line by themselves limits CPP's granularity somewhat. For C-like languages, CPP provides very fine-grained variability since practically any syntactic element can be isolated on a line with directives added above and below. For other types of artifacts, however, this could be an issue.

The `#ifdef` directive provides a very straightforward way to capture and conditionally include features in CPP. We can simply use a different macro name for each feature and wrap all code corresponding to a particular feature in appropriate `#ifdef` statements. The example in Figure 3 demonstrates this approach. In this example, a class representing a generic buffer contains two optional features—one which logs the value of the buffer each time it is changed, and one that provides a limited undo capability, to restore the previous value of the buffer.

Clearly, the notational overhead of CPP's syntax completely overwhelms the code we are actually interested in, and even this simple example is nearly incomprehensible. This is just one of a few well-known and long-recognized problems with CPP, despite its continued use [30].

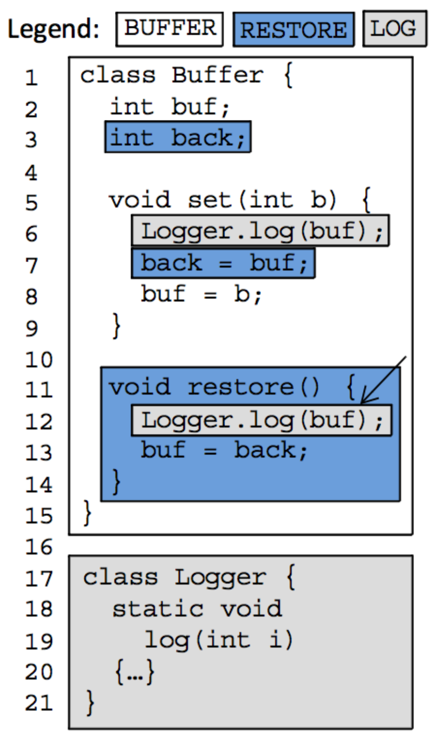
Other issues with CPP mostly revolve around its lack of structure. We have used macros to represent features, but this is obviously not captured explicitly in the CPP code anywhere. Nor is there any representation of relationships between these features. We must also consider that some conditional expressions will not just depend on a single macro, but on arbitrary integer expressions containing macros. All of this makes documentation of how macros are used and how they are related essential for understanding larger programs. Unfortunately, such documentation rarely exists.

Finally, an issue that CPP shares with any truly generic annotative approach is that it is difficult to make any static guarantees about how an artifact is varied. For example, it would be nice to definitively say that all variations of an artifact are syntactically valid. With no constraints on where and how variation can occur, this is not possible without examining every variation or performing some other complex analysis.

All of these issues make CPP difficult to understand and error-prone as a tool for variation management. The tool described next solves many of these issues by providing a less intrusive annotation mechanism, a simple feature representation, and tying variation to nodes in an AST.

## 5.2 CIDE

The Colored IDE, or CIDE [17], is a graphical tool for creating and managing features. Users can edit code in CIDE, as in a regular IDE, but can also create features, assign code to features, select and deselect features, and export code corresponding to



```
class Buffer {
  int buf;
  int back;

  void set(int b) {
    Logger.log(buf);
    back = buf;
    buf = b;
  }

  void restore() {
    Logger.log(buf);
    buf = back;
  }
}

class Logger {
  static void
    log(int i)
  {…}
}
```

**Figure 4:** Buffer with optional logging and restore features in CIDE (image from [21]).

the current feature selection. Feature annotations are shown in different colors—each feature is assigned a color, and code corresponding to a feature is highlighted in its color. Figure 4 demonstrates this with our logged and restorable buffer example.

In addition to being significantly easier to read and understand than the CPP representation, CIDE provides interactions which further support comprehensibility. If a feature is deselected its code is removed from the display (leaving a small marker in its place). This provides a *virtual separation of concerns*—although the code corresponding to a feature is not modularized, it can still be removed from consideration when trying to understand other parts of the program. CIDE also provides navigation operations for moving between pieces of code associated with a common feature.

CIDE's annotation model also differs from CPP's in that it constrains feature assignment to nodes in the AST of the underlying language. This avoids some of the structural issues of CPP and other generic annotated approaches, but does so at the cost of generality—CIDE must be able to parse the underlying language in order to annotate it. To annotate a node with a feature, the user highlights the corresponding code in the interface (the selection snaps to correspond with an AST node, accordingly), then selects the feature to assign it to. Any element in the AST can be annotated, providing very fine-grained variation.

Features in CIDE are either present or not, and no mechanism is provided for describing additional relationships between features. A more advanced feature model could easily be incorporated, however, perhaps incorporating user interface elements from the GUI translations of feature models in Section 6.3. Some work has been done on deriving feature models from CIDE annotations [21].

### 5.3 XVCL

The *XML-based Variant Configuration Language* (XVCL) [35, 36] is a hybrid annotative and compositional variation management system. It aims for the generality, flexibility and fine-grained granularity of annotative approaches, but also supports true separation of concerns like compositional approaches. XVCL provides several commands, in the form of XML tags, which can be added to arbitrary artifacts. It is unique in providing both a mechanism for in-place variation and mechanisms for distributed variations.

In-place variation is specified by the `<select>` command, which contains an arbitrary number of `<option>` blocks and an optional `<otherwise>` block. Option and otherwise blocks can contain arbitrary text. Each selection is associated with a named variable, and each option is associated with a potential value of that variable. When a selection is encountered, the value of the corresponding variable is used to determine which option's code is included. If the value doesn't match any options, the code in the otherwise block is included, if present. This representation of in-place variation is somewhere between CPP and the choice calculus. Selections, variables, and values correspond roughly to choices, dimension names, and tags, but unlike in the choice calculus, the completeness of selections is not enforced. Also, as in CPP, the values of variables can change at selection time, i.e. there are XVCL commands which change the value of variables. In XVCL, in-place variation is very flexible, but also potentially error prone.

Distributed variation is provided through the `<break>` and `<insert>` commands, which are very similar to AOP join points and advice, respectively. A break is just a named point at which code can be inserted. Since breaks can be added anywhere, they are more flexible than join points which are limited to well-defined places in the control flow. But adding a break also requires modifying the source code, whereas code can be added to join points in an existing program without modification. An insert is essentially identical to advice, specifying a block of code to insert before, after, or in place of a break.

XVCL also provides mechanisms for declaring and copying named chunks of code to facilitate reuse.

## 6 Feature Models

Until now, we have focused almost exclusively on how representations capture features, but another important part of variation management and SPL development is understanding how these features are related. Which features form a dimension of variation? Which combinations of features produce valid or desirable programs? Are there constraints on the order of feature application? In some cases this information is encoded directly in the feature representation; for example, the choice calculus provides explicit dimension declarations. In other cases, this information can be at least partially derived by analyzing the feature representation. The most common and general solution, however, is to supplement the feature representation with a higher level description of feature relationships.

*Feature models* describe relationships between features and act as constraints on the set of all possible feature combinations. Given $n$ features and no relationships between them, each feature can either be included or not, giving a total of $2^n$ variations. If we consider all possible orderings of

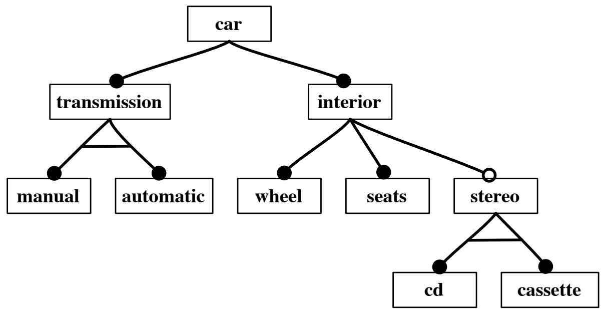


**Figure 5:** Partial feature diagram for a car.

all possible combinations of features, the number of variations grows substantially larger. Fortunately, the number of valid or desirable variations is usually many orders of magnitude smaller. Feature models can help isolate these interesting cases and discard the invalid and uninteresting ones.

In this section we look at a representative sample of feature models. As an aside, note that in the literature, the term "feature model" sometimes refers only to the hierarchical feature models described in Section 6.1. We use the term more broadly to mean any model or language which describes relationships between features.

## 6.1 Hierarchical Models and Feature Diagrams

Many feature representations lead naturally to hierarchically structured feature dependencies. Compositional approaches based on stepwise refinement, in particular, are strongly hierarchical by design [6]. Users of these systems often rely on a variation of the hierarchical feature models and diagrams described in this subsection. Even when these models are not directly integrated into the system, they are used as design documents to help develop and understand feature-oriented software.

*Hierarchical feature models* are most naturally encoded as *feature diagrams* [16]. Feature diagrams are a tree-based visual notation for describing relations between features. An example feature diagram is given in Figure 5, describing the relationship of a few features from a feature-oriented design of a car. The following relations are encoded in this diagram:

- Children are *dependant* on their parents. A child cannot be included unless its parent is also included. For example, we cannot include the "cd" feature, unless the "stereo" feature is also included. The root feature is not dependent on any feature.
- Edges connected by horizontal lines establish *alternatives*. Only edges from a common parent can be connected as alternatives, and exactly one descendant from the set of alternative edges must be included, assuming the parent feature is also included. Thus, if the "transmission" feature is included in the example, we must also include *either* the "manual" or "automatic" feature.
- Children with solid dots are *mandatory*—if the parent of a mandatory feature is included, it must also be included (except as overridden by the alternative relationship, which takes precedence). Thus, an "interior" must have both a (steering-) "wheel" and "seats".
- Children with hollow dots are instead *optional*—if the parent of an optional feature is included, the feature *may* be included, or not. Features included in an alternative relationship cannot be marked optional. In the example, the "stereo" feature may optionally be included if the "interior" feature is included.

The original definition of feature diagrams in [16] also allows arbitrary supplemental relationships, written in plain text. For example, we might specify that if we choose an automatic transmission, then we must also include a stereo, a relationship that cannot be captured in this notation.

The notation above represents the common core of many related feature diagram notations. Schobbens et al. describe many extensions to feature diagrams that have been developed over the years, provide a generalized (but less usable) representation that subsumes most of these, and provide a formal semantics for the generalized notation [29].

## 6.2 Feature Algebras

While users of compositional variation systems often rely on feature diagrams to design and understand their code, the systems themselves instead rely on much simpler *feature algebras*.

The feature algebra used in GenVoca is extremely simple—refinements are represented as functions, programs as constants, and every variation of interest is explicitly specified by successively applying functions to constants [5].

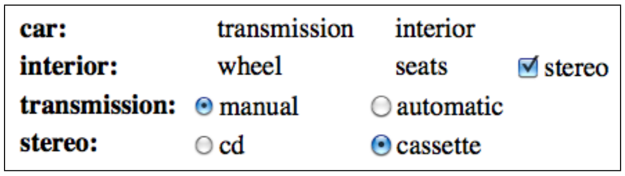
The algebra used by AHEAD is only slightly more advanced. In AHEAD, constants and refinements are recursively defined as lists of smaller constants and refinements, as described in Section 3.3. Function composition is then defined as a component-wise composition of elements—composing refinements like function composition, and taking the union of constants. From here the user proceeds as in GenVoca, providing explicit definitions of every desired variant [6]. Apel et al. provide a fundamentally similar, but generalized model that can accommodate other feature representations like aspects [3].

The feature algebra created by Höfner et al. [15] is a much more theoretically intensive, and fundamentally more general approach, more in the spirit of the choice calculus than other representations seen so far. This algebra is based on the concept of *idempotent semirings*, which is a fancy way of saying the formalism operates over a set of values, contains two operations for composing values, and two special values, all of which obey a certain set of laws. In the context of combining features to form product families, the two operations correspond to a choice between two features/product lines (“or”), and the co-occurrence of two features/product lines (“and”).

The authors demonstrate the algebra to be very powerful and general compared to other feature models, and to better support theoretical results related to correctness and reusability. Compared to the choice calculus, the feature algebra is quite abstract—containing no mechanism for representing code-level variation, and not tied to any particular feature structure (the choice calculus is a fundamentally dimension-oriented approach).

```
car          := transmission interior
interior     := wheel seats [stereo]
transmission := manual | automatic
stereo       := cd | cassette
```

**Figure 6:** Car feature diagram rendered as a grammar.



|  |  |  |  |
|---|---|---|---|
| **car:** | transmission | interior | |
| **interior:** | wheel | seats | [✓ stereo] |
| **transmission:** | [✓ manual] | automatic | |
| **stereo:** | cd | [✓ cassette] | |

**Figure 7:** Car feature diagram rendered as a GUI.

### 6.3 Other Feature Models

There are many other potential feature model representations. Batory provides straightforward transformations of feature diagrams into many other representations, including various types of grammars, propositional formulas, and even as a simple GUI representation [4]. Examples of a couple of these transformations on the car feature model from Figure 5 are shown in Figures 6 and 7.

A class of restricted grammars called *iterative tree grammars*, of which the example in Figure 6 is an instance, have the same expressiveness as feature diagrams. Internal nodes in the feature diagram appear as non-terminals on the left with their children on the right. *And*-branching (the default type of branching, with mandatory children) is equivalent to juxtaposition on the right, while *alternative*-branching is equivalent to disjunction. Optional features are indicated in square brackets, and are correspondingly optional in the grammar.

In the GUI representation, alternatives are represented as groups of radio buttons while optional features have check-boxes. These types of simple GUIs for feature selection can be automatically generated from a feature model [1]. This could be useful in an extension to CIDE, which is very good at representing features, but provides essentially no higher-level feature model.

## 7 Conclusions

Representations of software variation can be studied and categorized along many dimensions. We can separate the problem space by recognizing distinctions between long-term variations and short-term, multi-dimensional variations and single-dimensional ones. In this work, we focus on just one of the quadrants created by this partitioning, but our eventual goal is to support all four. This will require exploring in more depth the existing work in these other three areas. Previously, we have devoted some effort to understanding the quadrant of long-term, single-dimension variations, the area covered by revision control and SCM. In future work we will have to see what, if any, work exists in the context of short-term variations.

We can also separate the solution space along several axes. For the area of the problem space explored here, we have observed that there are two levels involved in representing features. The lower level describes which code corresponds to what features, and how to put it all together. The higher level describes relationships between features and how these features combine to form variations. The lower level can be further separated into compositional approaches vs. annotative approaches, which have distinctly different goals and thus capture very different notions of features. And, finally, compositional approaches can be separated into FOP- and AOP-based approaches, whose strengths are largely complementary and which recent research is beginning to combine to great effect.

In developing the choice calculus and its associated theory, it will be useful to keep these categories in mind. They are a means of abstraction, helping us to understand the scope of the problem, while freeing us from continually descending into the details of particular representations.

## References

[1] M. Antkiewicz and K. Czarnecki. FeaturePlugin: Feature Modeling Plug-In for Eclipse. In *OOPSLA Work. on Eclipse Technology eXchange*, pages 67–72. ACM Press, 2004.

[2] S. Apel, T. Leich, and G. Saake. Aspectual feature modules. *IEEE Trans. on Software Engineering*, 34(2):162–180, 2008.

[3] S. Apel, C. Lengauer, B. Möller, and C. Kästner. An Algebra for Features and Feature Composition. In *Int. Conf. on Algebraic Methodology and Software Technology*, volume 5140 of *LNCS*, pages 36–50. Springer-Verlang, 2008.

[4] D. Batory. Feature Models, Grammars, and Propositional Formulas. In *Int. Software Product Line Conf.*, volume 3714 of *LNCS*, pages 7–20. Springer-Verlang, 2005.

[5] D. Batory and S. O'Malley. The Design and Implementation of Hierarchical Software Systems with Reusable Components. *ACM Trans. on Software Engineering and Methodology*, 1(4):355–398, 1992.

[6] D. Batory, J. N. Sarvela, and A. Rauschmayer. Scaling Step-Wise Refinement. *IEEE Trans. on Software Engineering*, 30(6):355–371, 2004.

[7] J. Bosch, G. Florijn, D. Greefhorst, J. Kuusela, H. Obbink, and K. Pohl. Variability Issues in Software Product Lines. In *Int. Workshop on Software Product-Family Engineering*, volume 2290 of *LNCS*, pages 13–21. Springer-Verlang, 2001.

[8] G. Bracha and W. Cook. Mixin-Based Inheritance. In *ACM SIGPLAN Int. Conf. on Object-Oriented Programming, Systems, Languages, and Applications*, pages 303–311, 1990.

[9] G. Chastek, P. Donohoe, and J. McGregor. Formulation of a Production Strategy for a Software Product Line. Technical Report CMU/SEI-2009-TN-025, Software Engineering Institute, Carnegie Mellon University, Aug. 2009.

[10] M. D. Ernst, G. J. Badros, and D. Notkin. An Empirical Analysis of C Preprocessor Use. *IEEE Trans. on Software Engineering*, 28(12):1146–1170, 2002.

[11] M. Erwig and E. Walkingshaw. The Choice Calculus: A Representation for Software Variation to Support a Variation Design Theory, 2009. Draft Paper. `http://eecs.oregonstate.edu/~erwig/papers/cc-draft.pdf`.

[12] J. Estublier, D. Leblang, A. van der Hoek, R. Conradi, G. Clemm, W. Tichy, and D. Wiborg-Weber. Impact of Software Engineering Research on the Practice of Software Configuration Management. *ACM Trans. on Software Engineering and Methodology*, 14(4):383–430, 2005.

[13] E. Gamma, R. Helm, R. Johnson, and J. Vlissides.

*Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, Reading, MA, 1995.

[14] GNU Project. *The C Preprocessor*. Free Software Foundation. `http://gcc.gnu.org/onlinedocs/cpp/`.

[15] P. Höfner, R. Khedri, and B. Möller. Feature Algebra. In *Int. Symp. on Formal Methods*, volume 4085 of *LNCS*, pages 300–315. Springer-Verlang, 2006.

[16] K. C. Kang, S. G. Cohen, J. A. Hess, W. E. Novak, and A. S. Peterson. Feature-Oriented Domain Analysis (FODA) Feasibility Study. Technical Report CMU/SEI-90-TR-21, Software Engineering Institute, Carnegie Mellon University, Nov. 1990.

[17] C. Kästner, S. Apel, and M. Kuhlemann. Granularity in Software Product Lines. In *IEEE Int. Conf. on Software Engineering*, pages 311–320, 2008.

[18] A. Kellens, K. Mens, J. Brichau, and K. Gybels. Managing the Evolution of Aspect-Oriented Software with Model-Based Pointcuts. In *European Conf. on Object-Oriented Programming*, volume 4067 of *LNCS*, pages 501–525. Springer-Verlang, 2006.

[19] G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, and W. G. Griswold. An Overview of AspectJ. In *European Conf. on Object-Oriented Programming*, volume 2072 of *LNCS*, pages 327–354. Springer-Verlang, 2001.

[20] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. V. Lopes, J.-M. Loingtier, and J. Irwin. Aspect-Oriented Programming. In *European Conf. on Object-Oriented Programming*, volume 1241 of *LNCS*, pages 220–242. Springer-Verlang, 1997.

[21] C. H. P. Kim, C. Kästner, and D. Batory. On the Modularity of Feature Interactions. In *Int. Conf. on Generative Programming and Component Engineering*, pages 19–23, 2008.

[22] C. W. Krueger. Variation Management for Software Production Lines. In *Int. Software Product Line Conf.*, volume 2379 of *LNCS*, pages 37–48. Springer-Verlang, 2002.

[23] M. Mezini and K. Ostermann. Variability Management with Feature-Oriented Programming and Aspects. *ACM SIGSOFT Software Engineering Notes*, 29(6):127–136, 2004.

[24] L. M. Northrop and P. C. Clements. *A Framework for Software Product Line Practice, Version 5.0*. Software Engineering Institute, Carnegie Mellon University, 2007. `http://www.sei.cmu.edu/productlines/frame_report/`.

[25] H. Ossher and P. Tarr. Hyper/J: Multi-Dimensional Separation of Concerns for Java. In *IEEE Int. Conf. on Software Engineering*, pages 734–737, 2000.

[26] D. L. Parnas. On the Design and Development of Program Families. *IEEE Trans. on Software Engineering*, 2(1):1–9, 1976.

[27] K. Pohl, G. Böckle, and F. van der Linden. *Software Product Line Engineering: Foundations, Principles, and Techniques*. Springer-Verlang, Berlin Heidelberg, 2005.

[28] M. J. Rochkind. The Source Code Control System. *IEEE Trans. on Software Engineering*, 1(4):364–370, 1975.

[29] P.-Y. Schobbens, P. Heymans, and J.-C. Trigaux. Feature Diagrams: A Survey and a Formal Semantics. In *IEEE Int. Requirements Engineering Conf.*, pages 139–148, 2006.

[30] H. Spencer and G. Collyer. #ifdef Considered Harmful, or Portability Experience With C News. In *Proc. of the USENIX Summer Conf.*, pages 185–198, 1992.

[31] P. Szekely, P. Luo, and R. Neches. Facilitating the Exploration of Interface Design Alternatives: The HUMANOID Model of Interface Design. In *ACM SIGCHI Conf. on Human Factors in Computing Systems*, pages 507–515, 1992.

[32] P. Tarr, H. Ossher, W. Harrison, and S. M. Sutton Jr. N Degrees of Separation: Multi-Dimensional Separation of Concerns. In *IEEE Int. Conf. on Software Engineering*, pages 107–119, 1999.

[33] C. Walrad and D. Strom. The Importance of Branching Models in SCM. *Computer*, 35(9):31–38, 2002.

[34] N. Wirth. Program Development by Stepwise Refinement. *Comm. of the ACM*, 14(4):221–227, 1971.

[35] H. Zhang and S. Jarzabek. XVCL: A Mechanism for Handling Variants in Software Product Lines. *Science of Computer Programming*, 53(3):381–407, 2004.

[36] H. Zhang, S. Jarzabek, and S. M. Swe. XVCL Approach to Separating Concerns in Product Family Assets. In *Int. Conf. on Generative and Component-Based Software Engineering*, volume 2186 of *LNCS*, pages 36–47. Springer-Verlang, 2001.