Detecting Dependences and Interactions in Feature-Oriented Design

Sven Apel, Wolfgang Scholz, Christian Lengauer
University of Passau, Germany
{apel,scholz,lengauer}@fim.uni-passau.de

Christian Kästner
Philipps University Marburg, Germany
kaestner@Mathematik.Uni-Marburg.de

***Abstract*—*Feature-oriented software development* (*FOSD*) aims at the construction, customization, and synthesis of large-scale software systems. We propose a novel software design paradigm, called *feature-oriented design*, that takes the distinguishing characteristics of FOSD into account, especially the clean and consistent mapping between features and their implementations as well as the tendency of features to interact inadvertently. We extend the lightweight modeling language Alloy with support for feature-oriented design and call the extension *FeatureAlloy*. By means of an implementation and four case studies, we demonstrate how feature-oriented design with FeatureAlloy facilitates separation of concerns, variability, and reuse of models of individual features and helps defining and detecting semantic dependences and interactions between features.**

I. Introduction

The idea of *feature-oriented software development* (*FOSD*) is to decompose a software system in terms of the features it provides [1]. A *feature* is a unit of functionality that satisfies a requirement, represents a design decision, and provides a potential configuration option. Typically, from a set of features, many different software systems (a.k.a. *variants*) can be generated that share common features and differ in other features. The complete set of variants is also called a *software product line* [2].

Systematic software product-line development based on features has a number of benefits, among others, the ability to generate reliable and efficient software systems based on well-tested and verified software artifacts [2], [3]. So it is not surprising that the product-line paradigm has received considerable attention in research and industry.[1]

A distinguishing property of FOSD, compared to other product line engineering approaches [3], is the clean mapping between features and their implementations, which is achieved by expressive module and composition mechanisms [1]. The key idea is to implement each feature by a distinct *feature module* (or a well-defined set of alternative modules), thus establishing a clean mapping between problem space and solution space [2]. In FOSD, typically, a set of features and their relationships are identified in an analysis step, called the *domain analysis*, and then the features are implemented right away. That is, the current state of the art in FOSD research and practice does not take much advantage of contemporary design methods [1]. The reason may be the simplicity of the mapping from features to feature modules, which makes it unnecessary to model a *product line architecture* [3].

We propose a novel design paradigm, called *feature-oriented design*, that is tailored to the needs of FOSD. The idea is to take advantage of the clean mapping of features and their implementations and to concentrate on designing the structure and behavior of features as well as their dependences and interactions. We base our proposal of feature-oriented design on the lightweight but expressive modeling language *Alloy* [4]. We favor Alloy over other modeling languages, such as the unified modeling language (UML), because of its support of automatic reasoning. Alloy's automatic reasoning facilities are useful for detecting semantic dependences and interactions between features, which cause major problems in complex software systems [5] and are still a challenge for research and industry.

Alloy has no concept of a feature, so we had to extend it to be useful for FOSD. We call the extension *FeatureAlloy*. It is Alloy enhanced with support for collaboration-based design [6] and stepwise refinement [7]. With FeatureAlloy, a developer can model features separately and reason about their different combinations. We demonstrate by means of an implementation and four case studies that feature-oriented design with FeatureAlloy has several benefits:

- Feature-oriented design fills the gap between domain analysis and implementation. It allows developers to decompose a design in terms of features, thus facilitating *separation of concerns*, *variability*, and *reuse*.
- With FeatureAlloy, a developer can *express and detect dependences* between features at the semantic level, not only at the structural level.
- FeatureAlloy simplifies the *feature-interaction problem* [5] by providing support for the automatic detection of (certain kinds of) feature interactions (Sec. V).

Especially, the latter two mark notable improvements over previous work with regard to reliability.

II. Background

To lay a foundation for the subsequent sections, we explain the role of collaborations and refinement in FOSD and introduce basic concepts of Alloy.

[1] See the "Product Line Hall of Fame" for successful applications of product line technology in companies such as Boeing, Hewlett-Packard, and Philips: http://splc.net/fame.html.

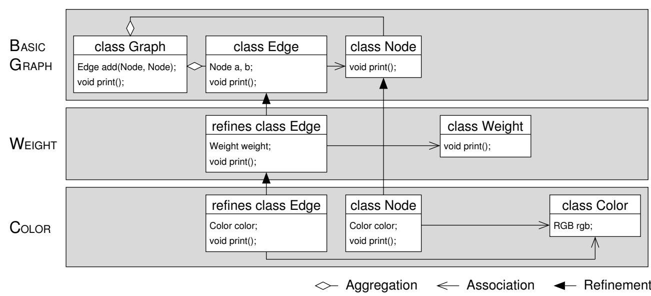

Fig. 1. Collaboration-based design of a simple graph structure.

A. Collaborations and Refinement

A popular technique for decomposing feature-oriented systems is *collaboration-based design* [6]. In Figure 1, we show a sample collaboration-based design of a graph library. A *collaboration* is a set of program elements that cooperate systematically to implement a feature. A collaboration comprises typically multiple classes and even only fragments of classes. The top-most collaboration (BASICGRAPH) consists of three classes: Graph, Node, and Edge. It represents the basic graph structure or the base program. The middle collaboration (WEIGHT) *refines* the base program by introducing a new class Weight, refining the existing class Edge by adding a new field weight and overriding method print to alter its behavior. The bottom-most collaboration (COLOR) adds a new class Color and refines class Node and Edge by introducing a new field color and overriding method print to alter its behavior.

In FOSD, each collaboration implements a feature and is called a *feature module* [8]. Different combinations of feature modules satisfy different needs of customers or application scenarios. Figure 1 illustrates how features crosscut the given hierarchical (object-oriented) program structure. In contemporary FOSD tools, such as AHEAD [7], FeatureC++ [9], FeatureHouse [10], or Fuji [11], collaborations are represented by file-system directories, called *containment hierarchies*, and classes and their refinements are stored in files. Features are selected by name via command line parameters or graphical tools. Feature composition is implemented by superimposing and merging recursively the directories and files based on the user's feature selection. It has been shown that feature composition can be applied to software artifacts written in various languages [10]. For more details on the composition process we refer the reader to a recent survey on FOSD [1].

B. Alloy

Alloy is a lightweight, textual modeling language for software design [4]. It is based on relations and logic, but has an object-oriented look and feel. This may be one reason for its acceptance in academia and industry. Its simplicity and sound mathematical foundation allow tools such as the Alloy Analyzer[2] to reason about Alloy models automatically (e.g., to decide whether there are legal instances of a model or whether certain properties hold in a model).

[2]http://alloy.mit.edu/

Feature BASICGRAPH

```
1  module Graph
2  // a singleton graph contains multiple nodes
3  one sig Graph {
4    nodes: set Node
5  } {
6    Node in nodes
7  }
8  // each node has multiple incoming and outgoing edges
9  sig Node {
10   inEdges: set Edge, outEdges: set Edge, edges: set Edge
11 } {
12   edges = inEdges + outEdges
13 }
14 // each edge has a source and destination node
15 sig Edge {
16   src: one Node, dest: one Node
17 }
18 // defines proper connections between nodes and edges
19 fact prevNext {
20   all n: Node, e: Edge |
21     (n in e.src <=> e in n.outEdges) && (n in e.dest <=> e in n.inEdges)
22 }
23 // determines the number of reachable nodes (incl. the given node)
24 fun reachableNodes [n: Node] : Int {
25   #(n.^(edges.(src + dest)))
26 }
27 // property that a graph has no double edges
28 pred noDoubleEdges {
29   all e, e': Edge | e != e' => e.(src + dest) != e'.(src + dest)
30 }
31 // creates an instance without double edges
32 run noDoubleEdges for 5
33 // holds if the graph has no double edges
34 assert hasNoDoubleEdges {
35   !noDoubleEdges
36 }
37 // checks whether the graph has no double edges
38 check hasNoDoubleEdges for 5
```

Fig. 2. An Alloy model of a simple graph.

We explain the key aspects of Alloy by means of our running example: the graph model. Figure 2 contains a simple Alloy model of our graph.[3] The model is defined in an Alloy module (keyword module) with name Graph (Line 1). The module contains three signatures (keyword sig) that represent the graph (Lines 3–7), nodes (Lines 9–13), and edges (Lines 15–17). In some sense, a signature is akin to an object-oriented class, but it is purely relational and thus more like a record. By means of additional constrains in a signature, expressed in algebra and logic, we can define, for example, that each model instance consists of only a single graph (modifier one; Line 3), each node has a set of incoming and outgoing edges (Line 10) whose union, denoted by field edges (Line 10), forms the entire set of edges (Line 12), and each edge has exactly one source and one destination node (Line 16).

With an Alloy fact (keyword fact), we define an axiom that holds for all model instances. In the graph model, we define that, for all nodes and edges, if a node is referred to by an edge as source node, the very edge is referred to by the node as outgoing edge—correspondingly for incoming edges (Lines 19–22).

Beside signatures and facts, Alloy supports the definition

[3]Our examples are written in Alloy 4 syntax and were tested with the Alloy Analyzer 4.1.10.

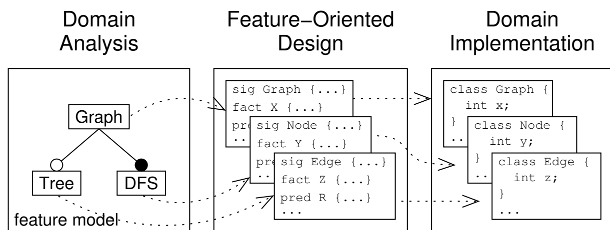

Fig. 3. The role of feature-oriented design in FOSD. (The feature model defines which feature combinations are valid.)

of functions and predicates (keywords fun and pred). We introduce a function for calculating the number of nodes that can be reached from a given node (Lines 24–26) and a predicate stating that there are no double edges between pairs of nodes (Lines 28–30). Functions and predicates can be used to compute and analyze properties or can be invoked in the Alloy Analyzer to determine the properties of a model.

The Alloy Analyzer can create or run example instances of a model for which a given predicate holds (keyword run). In our example, we create instances of graphs without double edges (Line 32). Furthermore, we can check whether a certain property holds. A property is expressed in the form of an assertion (keyword assert) and checked for a given scope (keyword check). For example, we can check whether no model instances exist that contain double edges (Lines 34–38). Counterexamples are presented to the developer as graphs of model elements.

The Alloy Analyzer is a bounded model checker.[4] It is able to examine a finite space of cases (in our example, setting the scope to five nodes and five edges via keyword for in Lines 32 and 38 was sufficient) but the analysis is performed fully automatically. It does not require test cases and, for a given scope, the results of the analysis are definitive. Typically, the space of cases examined is huge, even for a small scope, and Alloy's analysis offers a degree of coverage unattainable in testing [12].

III. Feature-Oriented Design

A. An Overview of Feature-Oriented Design

The basic idea of feature-oriented design is to make features explicit in the design and to use a modeling language that supports the concept of a feature. In Figure 3, we illustrate the role of feature-oriented design as a link between domain analysis and implementation as well as a clean (ideally one-to-one) mapping between features in all phases. Feature-oriented design comprises two steps: (1) modeling features separately and (2) reasoning about feature combinations.

Modeling Features Separately: A feature-oriented design consists of a set of model fragments, each of which corresponds to the portion of the design that concerns a feature. The individual fragments can be composed in different combinations using different composition techniques (e.g., superimposition [13]).

The benefits of modeling features in distinct units are:

- **Separation of concerns:** A feature-oriented design allows a developer to structure a software design along features, which facilitates understandability, maintainability, and evolvability.
- **Variability:** The individual features of a design can be composed in different combinations yielding different *generated* designs. This flexibility of composition (which cannot be attained with the standard Alloy module mechanism) allows a developer to tailor a software to the needs of a customer or application scenario.
- **Reuse:** Features can be reused in different design variants without replicating information, which is not possible with monolithic designs. Systematic reuse of well-tested or verified features can improve reliability.

Reasoning About Feature Combinations: Given a set of features that structure the design, a developer can reason about their dependences and interactions. Features may relate to other features in different ways (e.g., "feature A requires feature B and excludes feature C"). Typically, dependences between features are expressed in a feature model and common configuration tools disallow the generation of a software product from an invalid feature selection [1]. However, it has been observed that it is not uncommon for dependences and interactions to occur at the implementation level without any information in the feature model [14], [15]. This leads to the generation of incorrect target code and unexpected program behaviors [16]–[18].

In the past, type systems have been proposed to find a certain class of such implementation-level or *structural dependences* [14], [19], [20]. However, there is a class of dependences that cannot be detected at the type level, which we call *semantic dependences*. We show how FeatureAlloy can be used to model and detect semantic dependences automatically thus improving the reliability of the generated designs. Furthermore, semantic dependences are related to the *feature-interaction problem* [5], which we discuss in Section III-D.

B. FeatureAlloy

FeatureAlloy extends Alloy by three ingredients useful for FOSD: (1) collaboration-based design, (2) stepwise refinement, and (3) feature composition.

In a nutshell, FeatureAlloy follows the philosophy of contemporary FOSD languages and tools [1]. It represents each feature as a containment hierarchy, which encapsulates a collaboration of model elements (signatures, facts, etc.) that belong to a feature. Furthermore, FeatureAlloy supports the refinement of existing Alloy modules, signatures, facts, and so on by subsequent features without the need to modify existing model elements. Note that, like in the seminal work on feature interactions in telecommunication systems [5], a feature is not necessarily declaratively complete and a developer may have to combine it with a base program or with other features. That

[4] Actually, the Alloy Analyzer is a model finder. The difference to other model checkers is not relevant here and discussed elsewhere [12].

Feature TREE

```
1  refines module Graph
2  // property that the graph is connected
3  pred isConnected {
4    some n: Node | (Graph.nodes = n) || (Graph.nodes = n.^(edges.(src + dest)))
5  }
6  // property that the graph is acyclic
7  pred noCycles {
8    all n: Node | n not in (n.^(outEdges.dest) + n.^(inEdges.src))
9  }
10 // property that each node has one or no parent
11 pred loneParent {
12   all n: Node | lone n.inEdges
13 }
14 // defines that the graph is a tree
15 fact isTree {
16   noDoubleEdges && isConnected && noCycles && loneParent
17 }
```

Fig. 4. A refinement that imposes tree properties on graph instances.

Feature UNIQUEVALUES

```
1  refines module Graph
2  // adds a value to each node
3  refines sig Node {
4    val: one Int
5  }
6  // defines that node values are unique
7  fact uniqueValues {
8    all disj n, n': Node | n.val != n'.val
9  }
```

Fig. 5. A refinement that assigns unqiue values to nodes.

is, features are often increments in program functionality [7]. Finally, features are composed based on an external and declarative user's specification. A generator superimposes the model elements of the features involved and produces the final model of the system.

Module and Signature Refinement: We explain refinement in FeatureAlloy by means of the graph example, which was also our first case study. Suppose we add a new feature TREE to the graph model that ensures that every graph instance is a tree. In Figure 4, we show a corresponding refinement of module Graph by the addition of feature TREE. The refinement is declared using keyword refines. The semantics of module refinement is that the elements of a refinement are added to the refined module. This resembles the flattening semantics of mixin modules [21]. In our case, the refinement adds several predicates that define a tree (Lines 3–13) and a corresponding fact that states that these properties hold for every graph instance (Lines 15–17).

Furthermore, we add a feature UNIQUEVALUES, which assigns unique values to the nodes of a graph. In Figure 5, we show a corresponding refinement that refines signature Node by adding a new field val and that adds a fact defining that node values are unique. Notice the similarity of module and signature refinement. If there is already a field with the same name, the new definition of the signature refinement overrides the existing definition.

Finally, suppose we would like to add a feature BINARYTREE that defines that every tree is a binary tree. In Figure 6, we show a corresponding refinement that defines an assertion (Lines 3–8) that can be used to check whether the existing

Feature BINARYTREE

```
1  refines module Graph
2  // holds if the graph is a tree
3  assert isTree {
4    all e, e': Edge | e != e' => e.(src + dest) != e'.(src + dest)
5    some n: Node | (Graph.nodes = n) || (Graph.nodes = n.^(edges.(src + dest)))
6    all n: Node | n not in (n.^(outEdges.dest) + n.^(inEdges.src))
7    all n: Node | lone n.inEdges
8  }
9  // checks whether the graph is a tree
10 check isTree for 5
11 // defines that each graph is a binary tree
12 fact binaryTree {
13   all n: Node | #n.outEdges =< 2
14 }
```

Fig. 6. A refinement that defines that every tree is a binary tree.

graph model is a tree (Line 10) and that defines, based on this assumption, that all trees are binary trees (Lines 12–14). We explain in Section III-C that assertions play a key role in detecting semantic dependences with FeatureAlloy.

Generalizing Refinement: We generalize the concept of refinement by introducing capabilities to refine elements other than signatures, including facts or predicates. However, this is trickier than one would expect. A closer look at the structure of Alloy modules reveals that there are three kinds of model elements, each with different properties regarding refinement: (1) signatures; (2) facts, predicates, and functions; and (3) assertions.

First, a signature is refined by adding new fields, as already explained. Second, a fact, predicate, or function is refined by overriding.[5] That is, the content of a refining element (list of constraints or expressions) overrides the content of a refined element. The overriding element may refer to the overridden element by means of keyword original. Third, a feature is not allowed to replace an assertion of another feature, but it may refine it by adding constraints using keyword original. The reason for this design decision is that we use assertions to model semantic dependences and interactions, and features should not be able to alter the requirements of other features.

Feature Composition: Once a user has selected a set of features by specifying their names, a generator assembles all model elements of the features involved. Specifically, it proceeds recursively by taking the union of all elements and applying refinements to their base elements, which is much like composition in collaboration-based design (cf. Sec. II).

To summarize, FeatureAlloy's capabilities of making features explicit in the design (i.e., to separate features) improve (1) variability in that a developer can express variants of a design in terms of feature combinations and (2) reuse in that individual features can be used in different variants of a design. Although previous work provides related capabilities (see Sec. V), the combination of FOSD and Alloy provides a unique opportunity to discover semantic feature dependences and interactions, as we explain next.

[5]The refining and the refined element must have identical signatures.

C. Structural and Semantic Dependences

Historically, in FOSD, researchers have been assuming that all dependences between features are documented in a feature model, such that a user cannot generate invalid products. However, recently, Thaker et al. have shown that real product lines usually contain dependences between feature implementations that are not documented in the feature model and vice versa [14].[6] An undocumented dependence can lead to syntax or type errors when generating and compiling a software product; we call this a *structural dependence*.

We would like to draw attention to a class of dependences, which we call *semantic dependences*, that cannot be detected with existing tools for safe composition.[7] A semantic dependence is like a structural dependence, yet it occurs not at the level of syntax or types but appears in the form of a misbehavior at run time. That is, if a feature requires the presence of another feature due to a semantic dependence, but the other feature is not selected, then the final product has an incorrect behavior (although it is well-typed).

Structural dependences are problematic; they are hidden until a particular variant (which may be one out of millions) is compiled and appear then in the form of syntax or type errors. Semantic dependences are even more problematic. They are hidden in some variants until run time when a certain program state is reached, and they may be responsible for unhandled exceptions, segmentation faults, race conditions, and so on.

Let us illustrate the difference between structural and semantic dependences by means of our graph example. Suppose we add a feature that defines that all binary trees have the search tree property (i.e., for each node, all children in the left subtree have smaller values and all children in the right subtree have larger values). In Figure 7, we depict a corresponding refinement. It divides the set of edges into left-hand and right-hand edges (Lines 3–5) and introduces a fact stating the search tree property (Lines 7–13) via the two helper predicates (Lines 15–21).

There are two problems with feature SEARCHTREE. First, it depends on feature UNIQUEVALUES in that it refers to field val of signature Node (e.g., in Line 16). This is a structural dependence, which is detected when a user generates a variant with feature SEARCHTREE and without feature UNIQUEVALUES. Second, feature SEARCHTREE depends on feature BINARYTREE (which, in turn, depends on feature TREE). This dependence is a semantic dependence, which cannot be detected offhand by the Alloy Analyzer. The reason is that the search-tree property can be defined for general graphs, but this does not make sense (e.g., due to possible cycles there is no notion of a subtree) and the resulting graph model instances do not match our intention.

Structural dependences have been explored exhaustively in the past [14], [15], [18] and are outside the scope of this paper. We concentrate on semantic dependences, which are less explored (see Sec. V) and more challenging. A major contribution of our approach of feature-oriented design is to make semantic dependences explicit, such that they can be detected in the design phase, long before the compile and run time of the software system. Our choice of building Feature-Alloy on Alloy provides a unique opportunity to discover semantic dependences. We use the assertion mechanism and the automatic analysis of the Alloy Analyzer to model and detect semantic dependences. This is a notable improvement over previous work connecting modeling and FOSD (see Sec. V).

We illustrate the role of the assertion mechanism in Feature-Alloy by means of feature SEARCHTREE. Figure 8 extends the listing of Figure 7 by defining an assertion that we use to check the tree and binary tree properties of graph instances (Lines 3–10). The Alloy Analyzer presents a counterexample when feature SEARCHTREE is present but BINARYTREE is not, thereby indicating an unsatisfied dependence. The remaining code of feature SEARCHTREE (beginning at Line 12) is similar to the original feature shown in Figure 7 and omitted for brevity. The general pattern of modeling and detecting semantic dependences is (1) to define an assertion that specifies the properties that must hold for a given feature

Feature SEARCHTREE

```
1  refines module Graph
2  // divides the set of edges into left and right edges
3  sig LeftEdge, RightEdge extends Edge { } {
4    LeftEdge + RightEdge = Edge
5  }
6  // defines that the graph is a search tree
7  fact searchTree {
8    all n: Node | (#n.outEdges = 2) =>
9      (some n.outEdges & LeftEdge && some n.outEdges & RightEdge)
10   all n: Node | all l: LeftEdge | all r: RightEdge |
11     (l in n.outEdges => (validLeftSubTree [l.dest.∗(outEdges.dest), n])) &&
12     (r in n.outEdges => (validRightSubTree [r.dest.∗(outEdges.dest), n]))
13 }
14 // search tree property for the left subtree
15 pred validLeftSubTree[children: Node, parent: one Node] {
16   all child : Node | child in children => child.val < parent.val
17 }
18 // search tree property for the right subtree
19 pred validRightSubTree[children: Node, parent: one Node] {
20   all child : Node | child in children => child.val > parent.val
21 }
```

Fig. 7. A refinement that defines that every binary tree is a search tree.

Feature SEARCHTREE

```
1  refines module Graph
2  // holds if the graph is a binary tree
3  assert isBinaryTree {
4    all e: Edge | e.src != e.dest
5    some n: Node | (Graph.nodes = n) || (Graph.nodes = n.^(edges.(src + dest)))
6    all n: Node | n not in (n.^(outEdges.dest) + n.^(inEdges.src))
7    all n: Node | (lone n.inEdges) && (#n.outEdges =< 2)
8  }
9  // checks whether the graph is a binary tree
10 check isBinaryTree for 5
11 // the remaining code is taken from Figure 7
12 sig LeftEdge, RightEdge extends Edge ...
13 ...
```

Fig. 8. Using an assertion to check whether a graph is a binary tree.

[6]It has been argued that some dependences are intrinsically implementation-specific and should not be included in the feature model [15].

[7]Note that there is some related, theoretical work on model checking product lines, which we discuss in Section V.

to operate correctly and (2) to check this assertion after feature composition for counterexamples. If there is a counterexample, a dependence has not been satisfied (i.e., a required feature is not present). We call this idiom Requires Idiom.

Applying the Requires Idiom, we can use the Alloy Analyzer to ensure that a feature composition does respect all semantic dependences between the features involved, without relying on 'nominal' information of the feature model (e.g., 'SEARCHTREE requires BINARYTREE'). The Requires Idiom is a way to define a semantic constraints between features that provide enough information to be checked automatically.

The Requires Idiom is very flexible in that it does not rely on syntactic information. The fact that a graph is a binary tree can be expressed in many different ways. The Requires Idiom can be used without knowing how the property is defined; it simply states which properties the resulting model has to have. Alternative variants of features TREE and BINARYTREE can be used with feature SEARCHTREE, as long as the resulting graph has the binary tree property, as defined in Figure 8.

Note that, in our example, some dependences could have been structural, but this is not generally the case, especially not in a distributed feature composition scenario [22]. The ability to model requirements declaratively with FeatureAlloy is useful in such a scenario. Often, a developer does not know which other features will be added to a system. So, the developer has to define the constraints that put a feature to work solely on the basis of the known base system and on its own semantics (e.g., without a proper feature model). FeatureAlloy supports precisely this process and helps to detect unsatisfied dependences early in the software life cycle.

D. Feature Interactions

The feature-interaction problem is related to feature dependences. Two features interact if their combined presence leads to misbehavior, whereas their mutually exclusive presences do not. In such a situation, additional code is necessary to adjust the structure or behavior of one or both features to let them coexist properly. The additional code is called a *derivative* [18] or *lifter* [16]. Some FOSD configuration tools [18] ensure that, depending on the feature selection, the appropriate derivatives are selected.

We distinguish between structural and semantic interactions. A structural interaction is caused by one or more structural dependences, a semantic interaction by one or more semantic dependences. Again, we concentrate on the more interesting semantic interactions; structural interactions have been explored in the past [14]–[18], and our work on FeatureAlloy adopts established solutions for Alloy, but does not add anything new to this problem.

Since there are no straightforward semantic feature interactions in our graph example, we illustrate semantic feature interactions by means of the classic example of a phone system with the two features CALLFORWARDING and CALLWAITING. Feature CALLWAITING allows one call to be suspended while a second call is answered. Feature CALLFORWARDING enables a customer to specify a secondary phone number to which additional calls are being forwarded when the phone is busy. If both features are present and a call comes in while another is active, the phone system has to decide whether the call should be forwarded or the user should be notified that another call has arrived. In the worst case, the system behaves or terminates erroneously.

In Figure 9, we show an excerpt of the design of a phone system that we developed with FeatureAlloy. It consists of the three features BASICPHONE, CALLFORWARDING, and CALLWAITING. The latter two contain assertions (Lines 19–24 and 37–44) that we use to check whether the features operate correctly, which is the case when each feature is used without the other. When the latter two features are selected, checking one or both assertions produces counterexamples that indicate a feature interaction. Basically, the assertion is used to check whether a call is forwarded or suspended properly and the phone system is in a valid state.

Feature BASICPHONE

```
1  module Phone
2  // models a phone system
3  sig Phone {
4    currentState: one State, ...
5  }
6  // models the states of a phone system
7  abstract sig State {}
8  one sig Idle, Busy extends State {} ...
9  // models the state transition for incoming calls
10 pred incomingCall [in: Call, disj p, p': Phone] { ... }
```

Feature CALLFORWARDING

```
11 module Phone
12 // adds a field to forward a call
13 refines sig Phone {
14   forward: lone Phone
15 }
16 // overrides the state transition to forward calls
17 refines pred incomingCall [in: Call, disj p, p': Phone] { ... }
18 // holds if a call is forwarded correctly
19 assert isForwarded {
20   all disj phone, phone': Phone | all inCall: Call |
21     (incomingCall [inCall, phone, phone']) =>
22       ((phone.currentState = Idle <=> no phone'.forward) &&
23       (phone.currentState = Busy <=> one phone'.forward))
24 }
25 // checks whether a call is forwarded correctly
26 check isForwarded for 5
```

Feature CALLWAITING

```
27 module Phone
28 // adds a field to refer to a waiting call
29 refines sig Phone {
30   waitingCall: set Call
31 } ...
32 // adds a new state for suspended phones
33 one sig Suspended extends State {}
34 // overrides the state transition to suspend busy phones
35 refines pred incomingCall [in: Call, disj p, p': Phone] { ... }
36 // holds if a busy phone is suspended correctly
37 assert isSuspended {
38   all disj phone, phone': Phone | all inCall: Call |
39     (incomingCall [inCall, phone, phone']) =>
40       ((phone.currentState = Idle <=> no phone'.waitingCall) &&
41       (phone.currentState = Busy <=> one phone'.waitingCall) &&
42       (phone.currentState = Suspended <=>
43         (some phone.waitingCall && some phone'.waitingCall)))
44 }
45 // checks whether a busy phone is suspended correctly
46 check isSuspended for 5
```

Fig. 9. A basic phone system and two features whose interaction is detected by assertions.

This example illustrates an important difference between semantic interactions and semantic dependences. Modeling a semantic dependence, we use an assertion to state which properties defined by other features must hold so that a given feature works correctly. Modeling a semantic interaction, we use an assertion to define which properties of a given feature must not be altered by other features to let the given feature work correctly. To distinguish the two cases, henceforth, we refer to the first as the *Requires Idiom* and to the second as the Excludes Idiom, since the first reveals situations in which something is missing and the second reveals situations in which something is too much.

A distinguishing property of FeatureAlloy is that it enables us not only to model and detect semantic interactions, but also to resolve them without modifying existing model elements. In Figure 10, we show an excerpt of two derivatives that adjust the combined behavior of CALLFORWARDING and CALLWAITING. Essentially, the first derivative eliminates the interference of CALLWAITING and the second the interference of CALLFORWARDING, without changing the features. In practice, the two derivatives could be merged to a single derivative that resolves the interaction based on a parameter.

Derivative CALLFORWARDING + CALLWAITING

```
1  refines module Phone
2  // disables the interfering effects of CallWaiting
3  fact disableWaiting {
4    Suspended not in Phone.currentState
5    all phone: Phone |
6      (phone.currentState = Idle || phone.currentState = Busy) =>
7        no phone.waitingCall
8  }
```

Derivative CALLWAITING + CALLFORWARDING

```
9  refines module Phone
10 // disables the interfering effects of CallForwarding
11 fact disableForwarding {
12   all phone: Phone | no phone.forward
13 }
```

Fig. 10. Two derivatives that resolve the interaction between CALLFORWARDING and CALLWAITING.

To summarize, FeatureAlloy can be used to model and detect feature interactions modularly. It allows a developer to specify the properties that have to hold when a particular feature is present in a system, without requiring knowledge about other features (e.g., in the form of a feature model or code). Our approach can detect feature interactions solely on the basis of the individual features' specification. This ability is especially of interest in systems in which features are combined that have been developed independently, such as in distributed feature composition [22]. FeatureAlloy enables a developer to resolve interactions using derivatives, which are themselves cohesive and composable units. This way, the resolution code does not pollute the base code, which would decrease variability and reusability.

IV. IMPLEMENTATION & CASE STUDIES

We have implemented FeatureAlloy on top of the FeatureHouse tool suite [10]. FeatureHouse is a general framework for FOSD, into which languages can be plugged to be enriched with support for features and feature composition. We plugged in Alloy and implemented the syntax and refinement rules explained in the previous sections, modulo some minor deviations.[8] In our case studies, we have used the Alloy Analyzer to model individual features (e.g., taking advantage of syntax checking and highlighting), FeatureHouse to compose FeatureAlloy designs, and again the Alloy Analyzer to check assertions and visualize instances of feature compositions. Especially, the visualization and analysis capabilities are an improvement over previous approaches of connecting modeling and FOSD (see Sec. V). FeatureAlloy can be downloaded from the Web as part of the FeatureHouse distribution.[9]

Beside the graph case study (which we extended by further features such as BINARYSEARCH) and the phone system case study, we have conducted two further case studies, which can be downloaded as part of the FeatureHouse distribution. We focused particularly on FeatureAlloy's capabilities of separating features and of modeling and detecting semantic dependences and interactions. We meant not to conduct an empirical case study but to explore the potential benefits and drawbacks of our approach. We tried to keep the models as concise as possible and to capture only the essential aspects of the target systems, as is best practice in software design [12]. Nevertheless, we found several dependences and interactions and were able to model, detect, and resolve them with our approach.

Content-Addressable Network: A *content-addressable network* (*CAN*) is a protocol for data management and routing in large peer-to-peer networks [23]. The basic idea is that the network manages an n-dimensional key-value space. Each peer is responsible for a certain region of the space and knows his adjacent peers and their regions. If a data item is queried, a peer can decide whether the corresponding key (which is an n-tuple) is in its own region of the space or not. In the former case, the corresponding data item is returned and, in the latter case, the query is forwarded to the neighbor whose region is closest to the key.

We have used FeatureAlloy to model the key aspects of data management and routing in a CAN. Overall, we have decomposed the design into eight features that represent key design decisions in developing a CAN (e.g., routing and item retrieval) and advanced functionalities such as load measurement and the simulation of malicious peers. In our design, all features are well separated, we can generate tailored CAN models based on declarative specifications (e.g., with or without malicious peers), and we can reuse model fragments across different CAN models (e.g., routing is present in all variants).

While modeling and composing the features, we discovered a semantic dependence: load measurement requires the CAN's key-value space to consist of only a single partition. Otherwise, the load measurement does not calculate the average load

[8] For example, in the current version, FeatureAlloy infers automatically which model element overrides another element; keyword refines is not necessary and is omitted for symmetry with other feature-oriented languages [10].

[9] http://www.fosd.de/fh/

correctly since it relies on the standard routing mechanism, which cannot route between multiple partitions. We have made this dependence explicit by applying the Requires Idiom. So, when we select load measurement in a CAN that may have multiple partitions, we detect this error automatically with the Alloy Analyzer. Without FeatureAlloy, this dependence may go unnoticed, especially when it is not documented properly.

Furthermore, we found two semantic feature interactions. In particular, the load-measurement feature and the malicious-peers feature interact such that we had to adjust a function that calculates the overall number of data items. The reason is that malicious peers do not respond to queries even though they own data items of interest. So, to calculate the correct number of data items, we have to ignore malicious peers. A similar interaction occurs between the item-retrieval feature and the malicious-peers feature. We have applied the Excludes Idiom to make the two interactions explicit, and we have implemented two derivatives to resolve the interactions. FeatureAlloy helped in both cases: interactions are detected with assertions and derivatives are modeled in distinct units.

POSIX File System: In a further case study, we created and analyzed a model of a POSIX-compliant file system. The basic file-system model is a legacy design created by others.We decomposed it into five features representing fundamental operations such as create, move, and remove. Then, we added further features for block and partition management, inspired by Kang and Jackson [24], and for symbolic links. Applying feature-oriented–design principles, we were able to attain a similar degree of separation of concerns, variability, and reuse as in the CAN case study.

We found a semantic interaction between the list operation and symbolic links. Applying the Excludes Idiom, we state and check that a list operation does not 'jump' between file systems. We used a derivative to disallow the list operation to follow symbolic links and to introduce an extended list operation that may bridge file systems.

Interestingly, in this case study, we noticed a potential limitation of the Excludes Idiom. The background is that block management interacts with file system operations. File system operations such as remove change the file system structure. If there is block management, the mapping between index nodes and files has to be kept consistent. We achieve consistency by applying additional code (i.e., derivatives) that synchronize file-system changes with the underlying blocks. The problem is to detect the interactions between block management and file-system operations. A corresponding assertion has to be aware of both the changes the operations apply *and* the underlying block management. Hence, the assertion code it not modular in the sense that it can be assigned to a single feature. We pose this problem as an open research question: Can *all* interactions be detected by formulating assertions that are local to individual features?

V. Related Work

Several researchers proposed to make features or other aspects of a system explicit in the design (a discussion of work that targets source code is outside the scope of this paper). Fisler and Krishnamurthi emphasized the role of features in system design and verification [25]. Trujillo et al. decomposed domain-specific state-chart models into features and composed them in different combinations based on a user's feature selection [26]. Apel et al. compose model fragments of UML diagrams in a feature-oriented way [13]. Prehofer models features and feature interactions with partial state charts and composes them automatically based on a user's feature selection [27]. Similarly, in aspect-oriented modeling (AOM), different model fragments are "woven" using certain more or less explicit composition rules [28], [29]. All of these approaches rely on modeling languages that are not sufficient for automated reasoning and thus not able to handle feature dependences and interactions properly.

Jayaraman et al. use graph-based rewriting techniques to compose UML models belonging to different features [30]. The rewriting approach allows them to detect structural interactions between features. Instead, with FeatureAlloy, we concentrate on semantic interactions. Mostefaoui and Vachon translate UML models into Alloy models to detect design-level interactions in Aspect-UML models [31]. Our aim has been to bridge the gap between domain analysis and implementation in FOSD, which goes beyond their work. Furthermore, Alloy is much simpler than UML and closer to formal verification approaches, such that Alloy models are easier to check automatically than UML models.

Uzuncaova et al. use Alloy specifications to generate test inputs for compositions of features [32]. Like ordinary feature code, the specifications can be refined by subsequent features to alter or extend the test coverage. They aim at improving the speed of test generation based on stepwise refinement, whereas we aim at feature-oriented design and the detection of feature dependences and interactions.

Work on early aspects and aspect-oriented requirements engineering attempts to infer automatically mutual conflicts between requirements and inconsistencies between requirements and the system architecture [33], [34]. To this end, architectural description languages, natural-language processing, and machine-learning techniques are used to support automated reasoning. However, requirements and architectural specifications are more abstract than Alloy models and are useful in earlier development phases than feature-oriented design. Furthermore, Alloy is a very lightweight approach based on a sound mathematical foundation.

Li et al. proposed a technique for the verification of feature-oriented systems that proceeds in two steps: (1) features are verified modularly and (2) their composition is verified without the need of re-verifying the involved features [35]. The state machine models they use are quite spartan, mainly designed for formal argumentation, and not user-friendly enough to support a design at a proper level of abstraction.

Poppleton proposed an approach for feature-oriented specification [36]. He uses a formal specification language, called FeatureEvent-B, to specify and model the structure and behavior of a feature in isolation and to ensure the safe composition

of features without considering the implementation. However, FeatureEvent-B is quite abstract and only of limited use as a general modeling language for feature-oriented design.

Recently, some model-checking techniques have been proposed that take the variability of models in software product lines, especially state machines, into account and guarantee certain correctness properties for all products that can be generated [37]–[39]. State-machine models are appropriate for verification, but they are too abstract for all facets of practical software design.

Finally, there is a significant body of research in the telecommunications domain that explores the feature-interaction problem [5]. Most related is the work of Zave, who modeled, for example, the interoperation of addressing schemes in network systems with Alloy [40]. Similarly to our approach, she used assertions that express the requirements of features. However, her approach does not make features explicit and thus is not able to compose features and their derivatives flexibly. Feature-oriented design with FeatureAlloy draws a connection between software product lines and FOSD and the work on feature interactions in telecommunication systems.

VI. Discussion & Further Work

Feature-oriented design allows a developer to compose features in different combinations tailoring the design to specific needs, to reuse features in different design variants, and to explore potential feature interactions. Using Alloy or any other related modeling or specification language, we could not make features explicit in our case studies thus losing the benefits of a proper separation of features such as reuse, variability, and modular feature specification.

In our case studies, we found some semantic dependences between features. We applied the Requires Idiom to make the dependences explicit and to detect feature combinations that do not satisfy them. In this respect, FeatureAlloy combines the strengths of FOSD and Alloy. Without Alloy's automatic reasoning, semantic feature dependences may not be detected and resolved and may lead to erroneous behavior. So, our approach is an improvement over previous work on connecting UML and FOSD, which lacks a proper automatic analysis.

Furthermore, we found in our case studies some semantic feature interactions, which we modeled and detected using the Excludes Idiom and which we resolved by means of derivatives. An advantage of FeatureAlloy (beside automated reasoning) is that we can model derivatives in distinct units, without altering existing model elements, and we can apply the derivatives depending on a developer's feature selection. So, FeatureAlloy has the ability to modularize even the glue or derivative code that fixes undesired interactions. Without a feature-oriented decomposition, the derivative code would be entangled with the feature code, and the coupling between the features would hinder flexible feature composition [15].

We believe that our approach is of broader interest because our case studies help to understand the difference between a simple semantic dependence and a semantic interaction and their effects on software reliability. The former reflects a situation in which a feature requires properties of other features. The latter reflects a situation in which a feature interferes undesirably with (local) properties of other features.

A helpful property of the Requires Idiom and the Excludes Idiom is that they enable us to model the requirements and constraints of a feature purely and modularly in terms of the properties of the model instances the feature expects. They allow us to decouple features from the syntactic structures of other features, a property that is essential when the other features of a system are not known a priori.

Finally, we would like to comment on the practicality of our approach. First, as with all automatic verification procedures, it relies on the correctness of the input model in the sense that it contains the essential facts of the world in which we are interested. Proving this kind of correctness is impossible and a principle limitation of computer-aided verification [41]. Second, the Alloy Analyzer searches for counterexamples by bounded model checking. That is, there is always the possibility that a counterexample is going unnoticed because of an undersized scope. Based on our experience with the case studies, we agree with Jackson that, although correctness can ultimately not be proved, Alloy (and FeatureAlloy) offer a good trade-off between reliability and performance [12]. Third, model checking is a computationally expensive task. Although much progress has been made in the past, large systems can still not be verified in a reasonable time. Fortunately, design, in general, and feature-oriented design, in particular, aim at capturing the key aspects of a system and at identifying design flaws. That is, typically, the resulting models are orders of magnitude smaller than corresponding code bases.

There are three promising avenues of further research. First, we would like to enhance FeatureAlloy's capabilities to check features in isolation. This requires a syntactic and semantic interface mechanism. Earlier work on the modular verification of features [25], [35] proposed a similar mechanism, and it is interesting to draw a connection. Second, we would like to move the dependence and interaction analysis from after to before composition time. This is in line with the vision of safe composition of product lines to check a product line for incorrect products upfront, before the (possibly many) individual products have been generated [14], [19], [20], [42], [43]. Some theoretical approaches have previously applied this idea to the model checking of product lines [37]–[39]. Our work on FeatureAlloy lays an important, practical foundation for detecting semantic errors in product lines. Third, we would like to carry the results of the feature-oriented design phase over to the implementation phase. How can we take advantage of FeatureAlloy specifications for code generation and verification in the implementation phase?

Acknowledgment

Wolfgang Scholz is being funded by the German Research Foundation (DFG—AP 206/2-1). Kästner's work is supported in part by the European Union (ERC grant ScalPL #203099).

References

[1] S. Apel and C. Kästner, "An Overview of Feature-Oriented Software Development," *J. Object Technology (JOT)*, vol. 8, no. 5, pp. 49–84, 2009.

[2] K. Czarnecki and U. Eisenecker, *Generative Programming: Methods, Tools, and Applications*. Addison-Wesley, 2000.

[3] K. Pohl, G. Böckle, and F. van der Linden, *Software Product Line Engineering. Foundations, Principles, and Techniques*. Springer-Verlag, 2005.

[4] D. Jackson, "Alloy: A Lightweight Object Modelling Notation," *ACM Trans. Software Engineering and Methodology (TOSEM)*, vol. 11, no. 2, pp. 256–290, 2002.

[5] M. Calder, M. Kolberg, E. Magill, and S. Reiff-Marganiec, "Feature Interaction: A Critical Review and Considered Forecast," *Computer Networks: Int. J. Computer and Telecommunications Networking*, vol. 41, no. 1, pp. 115–141, 2003.

[6] Y. Smaragdakis and D. Batory, "Mixin Layers: An Object-Oriented Implementation Technique for Refinements and Collaboration-Based Designs," *ACM Trans. Software Engineering and Methodology (TOSEM)*, vol. 11, no. 2, pp. 215–255, 2002.

[7] D. Batory, J. Sarvela, and A. Rauschmayer, "Scaling Step-Wise Refinement," *IEEE Trans. Software Engineering (TSE)*, vol. 30, no. 6, pp. 355–371, 2004.

[8] S. Apel, T. Leich, and G. Saake, "Aspectual Feature Modules," *IEEE Trans. Software Engineering (TSE)*, vol. 34, no. 2, pp. 162–180, 2008.

[9] S. Apel, T. Leich, M. Rosenmüller, and G. Saake, "FeatureC++: On the Symbiosis of Feature-Oriented and Aspect-Oriented Programming," in *Proc. Int. Conf. Generative Programming and Component Engineering (GPCE)*, ser. LNCS, vol. 3676. Springer-Verlag, 2005, pp. 125–140.

[10] S. Apel, C. Kästner, and C. Lengauer, "FeatureHouse: Language-Independent, Automated Software Composition," in *Proc. Int. Conf. Software Engineering (ICSE)*. IEEE CS, 2009, pp. 221–231.

[11] S. Apel, S. Kolesnikov, J. Liebig, C. Kästner, M. Kuhlemann, and T. Leich, "Access Control in Feature-Oriented Programming," *Science of Computer Programming*, 2010, to appear in the Special Issue on Feature-Oriented Software Development.

[12] D. Jackson, *Software Abstractions: Logic, Language, and Analysis*. MIT Press, 2006.

[13] S. Apel, F. Janda, S. Trujillo, and C. Kästner, "Model Superimposition in Software Product Lines," in *Proc. Int. Conf. Model Transformation (ICMT)*, ser. LNCS, vol. 5563. Springer-Verlag, 2009, pp. 4–19.

[14] S. Thaker, D. Batory, D. Kitchin, and W. Cook, "Safe Composition of Product Lines," in *Proc. Int. Conf. Generative Programming and Component Engineering (GPCE)*. ACM Press, 2007, pp. 95–104.

[15] C. Kästner, S. Apel, S. ur Rahman, M. Rosenmüller, D. Batory, and G. Saake, "On the Impact of the Optional Feature Problem: Analysis and Case Studies," in *Proc. Int. Software Product Line Conference (SPLC)*. SEI, 2009, pp. 181–190.

[16] C. Prehofer, "Feature-Oriented Programming: A Fresh Look at Objects," in *Proc. Europ. Conf. Object-Oriented Programming (ECOOP)*, ser. LNCS, vol. 1241. Springer-Verlag, 1997, pp. 419–443.

[17] J. Liu, D. Batory, and S. Nedunuri, "Modeling Interactions in Feature-Oriented Designs," in *Proc. Int. Conf. Feature Interactions in Software and Communication Systems (ICFI)*. IOS Press, 2005, pp. 178–197.

[18] J. Liu, D. Batory, and C. Lengauer, "Feature-Oriented Refactoring of Legacy Applications," in *Proc. Int. Conf. Software Engineering (ICSE)*. ACM Press, 2006, pp. 112–121.

[19] B. Delaware, W. Cook, and D. Batory, "Fitting the Pieces Together: A Machine-Checked Model of Safe Composition," in *Proc. Int. Symp. Foundations of Software Engineering (FSE)*. ACM Press, 2009, pp. 243–252.

[20] S. Apel, C. Kästner, A. Größlinger, and C. Lengauer, "Type Safety for Feature-Oriented Product Lines," *Automated Software Engineering—An International Journal*, vol. 17, no. 3, pp. 251–300, 2010.

[21] G. Lagorio, M. Servetto, and E. Zucca, "Featherweight Jigsaw – A Minimal Core Calculus for Modular Composition of Classes," in *Proc. Europ. Conf. Object-Oriented Programming (ECOOP)*, ser. LNCS, vol. 5653. Springer-Verlag, 2009, pp. 244–268.

[22] M. Jackson and P. Zave, "Distributed Feature Composition: A Virtual Architecture for Telecommunications Services," *IEEE Trans. Software Engineering (TSE)*, vol. 24, no. 10, pp. 831–847, 1998.

[23] S. Ratnasamy, P. Francis, M. Handley, R. Karp, and S. Schenker, "A Scalable Content-Addressable Network," in *Proc. Int. Conf. Applications, Technologies, Architectures, and Protocols for Computer Communications (SIGCOMM)*. ACM Press, 2001, pp. 161–172.

[24] E. Kang and D. Jackson, "Formal Modeling and Analysis of a Flash Filesystem in Alloy," in *Proc. Int. Conf. Abstract State Machines, B and Z (ABZ)*, ser. LNCS, vol. 5238. Springer-Verlag, 2008, pp. 294–308.

[25] K. Fisler and S. Krishnamurthi, "Decomposing Verification Around End-User Features," in *Proc. Int. Conf. Verified Software: Theories, Tools, Experiments (VSTTE)*, ser. LNCS, vol. 4171. Springer-Verlag, 2008, pp. 74–81.

[26] S. Trujillo, D. Batory, and O. Díaz, "Feature Oriented Model Driven Development: A Case Study for Portlets," in *Proc. Int. Conf. Software Engineering (ICSE)*. IEEE CS, 2007, pp. 44–53.

[27] C. Prehofer, "Plug-and-Play Composition of Features and Feature Interactions with Statechart Diagrams," *Software and Systems Modeling (SoSyM)*, vol. 3, no. 3, pp. 221–234, 2004.

[28] S. Clarke and E. Baniassad, *Aspect-Oriented Analysis and Design. The Theme Approach*. Addison-Wesley, 2005.

[29] J.-M. Jezequel, "Model Driven Design and Aspect Weaving," *Software and Systems Modeling (SoSyM)*, vol. 7, no. 2, pp. 209–218, 2008.

[30] P. Jayaraman, J. Whittle, A. Elkhodary, and H. Gomaa, "Model Composition in Product Lines and Feature Interaction Detection Using Critical Pair Analysis," in *Proc. Int. Conf. Model Driven Engineering Languages and Systems (MODELS)*, ser. LNCS, vol. 4735. Springer-Verlag, 2007, pp. 151–165.

[31] F. Mostefaoui and J. Vachon, "Design-Level Detection of Interactions in Aspect-UML Models using Alloy," *J. Object Technology (JOT)*, vol. 6, no. 7, pp. 137–165, 2007.

[32] E. Uzuncaova, D. Garcia, S. Khurshid, and D. Batory, "Testing Software Product Lines Using Incremental Test Generation," in *Proc. Int. Symp. Software Reliability Engineering (ISSRE)*. IEEE CS, 2008, pp. 249–258.

[33] N. Weston, R. Chitchyan, and A. Rashid, "A Formal Approach to Semantic Composition of Aspect-Oriented Requirements," in *Proc. Int. Requirements Engineering Conference (RE)*. IEEE CS, 2008, pp. 173–182.

[34] C. Chavez, A. Garcia, T. Batista, M. Oliveira, C. Sant'Anna, and A. Rashid, "Composing Architectural Aspects Based on Style Semantics," in *Proc. Int. Conf. Aspect-Oriented Software Development (AOSD)*. ACM Press, 2009, pp. 111–122.

[35] H. Li, S. Krishnamurthi, and K. Fisler, "Modular Verification of Open Features Using Three-Valued Model Checking," *Automated Software Engineering*, vol. 12, no. 3, pp. 349–382, 2005.

[36] M. Poppleton, "Towards Feature-Oriented Specification and Development with Event-B," in *Proc. Int. Work. Conf. Requirements Engineering: Foundation for Software Quality (REFSQ)*, ser. LNCS. Springer-Verlag, 2007, vol. 4542, pp. 367–381.

[37] K. Lauenroth, S. Toehning, and K. Pohl, "Model Checking of Domain Artifacts in Product Line Engineering," in *Proc. Int. Conf. Automated Software Engineering (ASE)*. IEEE CS, 2009, pp. 269–280.

[38] A. Gruler, M. Leucker, and K. Scheidemann, "Calculating and Modeling Common Parts of Software Product Lines," in *Proc. Int. Software Product Line Conference (SPLC)*. IEEE CS, 2008, pp. 203–212.

[39] A. Classen, P. Heymans, P.-Y. Schobbens, A. Legay, and J.-F. Raskin, "Model Checking Lots of Systems: Efficient Verification of Temporal Properties in Software Product Lines," in *Proc. Int. Conf. Software Engineering (ICSE)*. ACM Press, 2010, pp. 335–344.

[40] P. Zave, "A Formal Model of Addressing for Interoperating Networks," in *Proc. Int. Symp. Formal Methods (FM)*, ser. LNCS, vol. 3582. Springer-Verlag, 2005, pp. 318–333.

[41] B. Smith, "The Limits of Correctness," *SIGCAS Comput. Soc.*, vol. 14,15, no. 1,2,3,4, pp. 18–26, 1985.

[42] K. Czarnecki and K. Pietroszek, "Verifying Feature-Based Model Templates Against Well-Formedness OCL Constraints," in *Proc. Int. Conf. Generative Programming and Component Engineering (GPCE)*. ACM Press, 2006, pp. 211–220.

[43] C. Kästner and S. Apel, "Type-Checking Software Product Lines – A Formal Approach," in *Proc. Int. Conf. Automated Software Engineering (ASE)*. IEEE CS, 2008, pp. 258–267.