Held in conjunction with

the 5th International Conference of Software Language Engineering (SLE’12) and

the 11th International Conference on Generative Programming and Component Engineering (GPCE’12)

sponsored by
Program Committee

Sven Apel (University of Passau, DE)  
Joanne Atlee (University of Waterloo, CA)  
Maider Azanza (University of the Basque Country, ES)  
Don Batory (University of Texas at Austin, US)  
Paulo Borba (University of Pernambuco, BR)  
Jan Bosch (Chalmers University of Technology, SE)  
Goetz Botterweck (Lero, IE)  
Manfred Broy (Technische Universität Munich, DE)  
Dave Clarke (Katholieke Universiteit Leuven, BE)  
Martin Erwig (Oregon State University, US)  
Kathi Fisler (Worcester Polytechnic Institute, US)  
Alessandro Garcia (PUC-Rio, BR)  
Jeff Gray (University of Alabama, US)  
Florian Heidenreich (Technische Universität Dresden, DE)  
Patrick Heymans (University of Namur, BE)  
Christian Kästner (University of Marburg, DE)  
Thomas Leich (Metop, DE)  
Christian Lengauer (University of Passau, DE)  
Malte Lochau (Technische Universität Braunschweig, DE)  
Christian Prehofer (Fraunhofer ESK, DE)  
Rick Rabiser (Johannes Kepler University, AT)  
Stephan Reiff-Marganiec (University of Leicester, UK)  
Bernhard Rumpe (RWTH Aachen, DE)  
Stefan Sobernig (Vienna Uni of Economics and Business, AT)  
Maurice Ter Beek (Consiglio Nazionale delle Ricerche, IT)  
Ken Turner (University of Stirling, UK)
Preface

Feature orientation is an emerging paradigm of software development. It supports the largely automatic generation of large software systems from a set of units of functionality called features. The key idea of feature-oriented software development (FOSD) is to emphasize the similarities of a family of software systems for a given application domain (e.g., database systems, banking software, text processing systems) with the goal of reusing software artifacts among the family members. Features distinguish different members of the family. A challenge in FOSD is that a feature does not map cleanly to an isolated module of code. Rather it may affect (“cut across”) many components and documents of a software system. Research on FOSD has shown that the concept of features pervades all phases of the software life cycle and requires a proper treatment in terms of analysis, design, and programming techniques, methods, languages, and tools, as well as formalisms and theory.

The primary goal of the 4th International Workshop on Feature-Oriented Software Development is to foster and strengthen the collaboration between the researchers who work in the field of FOSD or in the related fields of software product lines, service-oriented architecture, model-driven engineering and feature interactions. The focus of FOSD’12 will be on discussions, rather than on presenting technical content only. Both workshop days start with a keynote by leading researchers in FOSD. Mira Mezini will talk about programming language concepts for FOSD and Salvador Trujillo is going to share experiences in applying FOSD to offshore wind power and railways. These keynotes will be an excellent start up for discussions on historical perspectives, current issues, and visions of FOSD.
Keynotes

Programming Language Concepts for Feature-Oriented Software Development

Mira Mezini, Darmstadt University of Technology, Germany

Object-oriented concepts of classes, inheritance and subtype polymorphism are praised for supporting the design of software that is open for extensions but closed for modifications. Yet, they fail to properly support feature encapsulation and extensibility. This has motivated work on late bound classes, advanced module concepts, and aspect-oriented programming. In the talk, I will present some of the work I have been doing in this space, specifically related to virtual and dependent classes, aspect-oriented and event-driven programming and will discuss the usefulness of these concepts for supporting feature-oriented software development.

FOSD-Engineering beyond Code: Experiences from Offshore Wind Power and Railways

Salvador Trujillo, IKERLAN Research Centre, Spain

Feature-Oriented Software Development (FOSD) is a software product line paradigm where products result from composing a set of units of functionality called features. Code-centric approaches, where a product’s source code is produced from the automated composition of features, dominated FOSD in early stages. Recently, research on FOSD has shown that the concept of features pervades all phases of the software life cycle and requires a proper treatment in terms of analysis, design, and programming techniques, methods, languages, and tools, as well as formalisms and theory. This presentation revisits code composition approaches and looks at models as a mechanism to attain higher abstraction levels. This is necessary for FOSD to scale towards larger software and systems engineering and to broaden the scope of the FOSD engineering lifecycle from software to systems engineering. Software artifacts become just another piece of the entire system. These ideas are illustrated with our experience FOSD-engineering industrial systems in practice for offshore wind power and railways domains.
# Table of Contents

Toward Variability-Aware Testing ............................................. 1
Christian Kästner, Alexander von Rhein, Sebastian Erdweg, Jonas Pusch, Sven Apel, Tillmann Rendel, and Klaus Ostermann

Conditioned Model Slicing of Feature-Annotated State Machines ... 9
Jochen Kamischke, Malte Lochau, and Hauke Baller

Comparing Program Comprehension of Physically and Virtually Separated Concerns .................................................. 17
Janet Siegmund, Christian Kästner, Jörg Liebig, and Sven Apel

Object-Oriented Design in Feature-Oriented Programming ........... 25
Sven Schuster and Sandro Schulze

Architectural Variability Management in Multi-Layer Web Applications Through Feature Models ........................................... 29
Jose Garcia-Alonso, Javier Berrocal Olmeda, and Juan Manuel Murillo

Ensuring Well-formedness of Configured Domain Models in Model-driven Product Lines Based on Negative Variability ............ 37
Thomas Buchmann and Felix Schwägerl

Supporting Multiple Feature Binding Strategies in NX ............ 45
Stefan Sobernig, Gustaf Neumann, and Stephan Adelsberger

Safe Adaptation in Context-Aware Feature Models ............... 54
Fabiana G. Marinho, Rossana M. C. Andrade, Paulo A. S. Costa, Paulo H. M. Maia, Vania M. P. Vidal, Claudia Werner

Towards a Catalog of Variability Evolution Patterns: The Linux Kernel Case ................................................................. 62
Leonardo Passos, Krzysztof Czarnecki, and Andrzej Wasowski

Challenges in the Evolution of Model-Based Software Product Lines in the Automotive Domain ........................................ 70
Hannes Holdschick
Toward Variability-Aware Testing

Christian Kästner
Philips University Marburg

Alexander von Rhein
University of Passau

Sebastian Erdweg and
Jonas Pusch
Philips University Marburg

Sven Apel
University of Passau

Tillmann Rendel and
Klaus Ostermann
Philips University Marburg

ABSTRACT

We investigate how to execute a unit test for all products of a product line without generating each product in isolation in a brute-force fashion. Learning from variability-aware analyses, we (a) design and implement a variability-aware interpreter and, alternatively, (b) reencode variability of the product line to simulate the test cases with a model checker. The interpreter internally reasons about variability, executing paths not affected by variability only once for the whole product line. The model checker achieves similar results by reusing powerful off-the-shelf analyses. We experimented with a prototype implementation for each strategy. We compare both strategies and discuss trade-offs and future directions. In the long run, we aim at finding an efficient testing approach that can be applied to entire product lines with millions of products.

1. INTRODUCTION

Analysis of software product lines has attracted much attention by researchers [26]. The addressed key problem is that traditional analysis methods (type checking, static analysis, model checking, testing, and so forth) target only individual programs, whereas a product line with $n$ optional compile-time features gives rise to $O(2^n)$ distinct configurations, and thus $O(2^n)$ distinct products. Traditionally, obtaining an analysis result for the entire product line (e.g., whether every product is well typed) would require to analyze each product in isolation, in a brute-force fashion. Since a brute-force approach does not scale due to the huge configuration space, practitioners resort to sampling strategies [5, 20–22]; they analyze only a few products currently produced, they analyze a few randomly selected products, or they analyze a relatively small number of products selected by some coverage criterion, such as t-way feature coverage. However, sampling cannot yield reliable analysis results for the entire product line.

Recently, researchers have investigated alternative strategies to analyze entire product lines without looking at the generated code of each product. We call analyses following these strategies variability-aware analysis (or family-based analysis [26]), because they take the variability of the product-line implementation into account during analysis. Roughly speaking, the idea is to analyze a generator (the product-line implementation itself together with configuration knowledge) instead of analyzing the generated products. Variability-aware analysis exploits the fact that products in a product line typically are generated from a common code base and share a significant amount of common code [10, 22]. When using brute force or sampling, this common code is analyzed repeatedly. In contrast, variability-aware analyses usually perform analysis on common code only once, while only variable code that actually affects the analysis result causes additional effort.

Researchers have successfully developed variability-aware analyses for parsing, type checking, model checking, static analysis, and theorem proving (see Sec. 5). Although testing of product lines has received significant attention, researchers have concentrated on sampling strategies [5, 20, 21], on test suite reduction [15, 24], and on test generation [24, 28]. In all these approaches, though, individual tests are still executed on generated products, one by one. To the best of our knowledge, there is no notion of variability-aware test execution, where a test is run on an entire product line without generating individual products.

Our goal is to transfer experience from existing variability-aware analyses to product-line testing. We want to execute a test case (e.g., a unit test) in all configurations of a product line, without actually generating a product for each configuration. In this workshop paper, we explore early steps in this direction. In line with extended mechanisms used in variability-aware analyses, we build a variability-aware interpreter to execute a test case in all configurations of a product line in parallel (which resembles mixed concrete/symbolic execution). Additionally, we explore an alternative strategy based on variability encodings and off-the-shelf analysis tools, in our case, JavaPathfinder (JPF) [29] and the extension jjf-bdd [30].

Specifically, our contributions are: We generalize strategies to implement variability-aware analyses into white-box and black-box strategies, which was only implicit in prior work. We design and implement a variability-aware interpreter for a WHILE language (white box). We apply JPF for variability-aware testing (black box). Finally, while we cannot yet make claims about scalability to real-world problems, we discuss trade-offs and limitations, and we outline research directions.

We want to encourage researchers to investigate testing of whole product lines without the usual sampling strategies. We are still in an early exploration stage toward variability-aware testing. Here, we present initial ideas and early experiences with prototypes and cases studies. We appreciate any feedback and ideas.

2. VARIABILITY-AWARE ANALYSIS

Before we discuss test-case execution in product lines, we briefly introduce variability-aware analysis in general, from which we then adopt many concepts. We start with the general goal, outline how we represent variability, and discuss two common implementation...
strategies.

We can explain variability-aware analysis with the process pattern illustrated in Figure 1. Instead of repeatedly generating a product (Step 1) and analyzing each product with a traditional analysis (Step 2), we want to analyze the entire product line without generating individual products (Step 3). Variability-aware analysis should produce a result that describes the entire product line. The result explains in which configuration which specific property holds (e.g., “all configurations with feature FOO are ill typed, all other configurations are well typed”). From this analysis result, we are able to deduce the properties that we would establish for an individual product with the traditional analysis (Step 4a). Alternatively, by applying the traditional analysis in a brute-force fashion to all products, we could aggregate the individual properties to describe the entire product line (Step 4b). While the output should be equivalent, we expect the variability-aware analysis (Step 3) to be much faster than the brute-force strategy (repeating Steps 1, 2, and 4b). In this paper, we want to apply this concept also to testing.

2.1 Variability representation

To perform variability-aware analysis, we need a structural representation of the product-line implementation that contains all compile-time variability. In our work, we encode compile-time variability directly in abstract syntax trees (ASTs) with presence conditions. A presence condition is a propositional formula over features of the product line that yields true iff the AST element (i.e., the corresponding code fragment) should be included in the product for a given configuration.

We manage variability with two constructs, as illustrated with Scala code in Figure 2: First, program elements can be optional (Opt[T] for elements of type T). An optional element is guarded by a propositional presence condition, which is represented by type FeatureExpr. Second, type Cond[T] encodes conditional elements, that is, elements that differ between configurations. We have either one element (One[T]) or a choice between two elements (Choice[T]) depending on a presence condition. Since choices can be nested, we can express multiple alternative elements. For example, we can express that variable \( v \) has value 1 if feature \( X \) is selected, and value 0 if feature \( Y \) but not \( Z \) is selected, and value –1 in all other cases: \( v = \text{Choice}(X, \text{One}(1), \text{Choice}(Y = \neg Z, \text{One}(2), \text{One}(-1))) \). Optional elements are typically used inside lists when \( 0..n \) elements are supported (e.g., a list of optional statements can contain no, one, or multiple statements in each configuration), whereas conditional elements are used when exactly one element is required in each configuration (e.g., an assignment always has exactly one right-hand-side expression).

Using Opt and Cond, we can express variability directly in the declaration of abstract syntax, as illustrated with the WHILE language in Figure 3 (the WHILE language is a small but Turing-complete imperative language, standard in static-analysis research). To create an AST with variability from source code with #ifdef directives, we use our variability-aware TypeChef parser [14]. We show an example WHILE program that contains variability in the form of preprocessor directives and the corresponding AST with variability in Figure 4.

Based on our AST representation with variability, we can realize variability-aware analyses for entire product lines, including the interpreter we present in Section 3.

1 Our current implementation allows arbitrary propositional formulas in choice nodes and uses a SAT solver to reason about variability. Instead of choice trees, we could alternatively store lists of optional entries, or encode conditional values similar to Boolean decision diagrams, or experiment with other representations, such as the Choice calculus [11].
2.2 Granularity, locality, and sharing

When specifying the abstract syntax of a language, we can decide where to inject variability in the AST. We can support variability at different levels of granularity, for example, allow conditional expressions inside assignments or merely allow optional elements at the statement level. We can always replace a fine-grained variability representation with a coarse-grained one at the cost of replication [11]. Usually, fine-grained granularity facilitates more sharing—sharing which we can potentially exploit to reduce analysis effort.

A key insight for variability-aware analysis is that, in all analysis steps, we want to keep variability as local as possible, to facilitate as much sharing as possible. For example, it is usually more efficient to store a map from names to conditional values than to store conditional maps from names to values. If we want to change a value in a single configuration in a representation of type \( \text{Cond}[\text{Map}[A,B]] \), we would need to copy the entire map, whereas changing a value in representation \( \text{Map}[A,\text{Cond}[B]] \) has a local effect and preserves sharing for all other values.

2.3 White-box vs. black-box strategy

Researchers have explored different strategies for variability-aware analysis. We observed that two general implementation strategies emerge, which we call henceforth white-box and the black-box strategy. Note that these terms are orthogonal to white-box vs. black-box testing to describe tests with and without source code (we do only white-box testing), but they refer to how analysis is performed and implemented.

White-box strategy. One common strategy is to extend the internal algorithm and data structures of the analysis. The modified analysis works on a representation with explicit variability, such as the ASTs presented above. It reasons about variability in all steps of the analysis and keeps variability local. Since we need to understand and modify the internals of the analysis, we name the strategy the white-box strategy.

For example, most variability-aware type checkers described in the literature follow the white-box strategy [1, 7, 13, 25]. Such a variability-aware type checker takes an AST with explicit variability information and exploits variability during analysis. The type checker knows in which configurations (described by a presence condition) a method is declared, and may even reason about conditional types of an expression. The analysis returns a list of conditional type errors, describing exactly in which configurations each error occurs.

In a white-box strategy, we extend the analysis to reason about variability. We perform analysis on shared code only once and only split analysis where variability actually occurs locally (late splitting). Also, when the analysis yields the same result in different configurations, the remaining analysis may be performed only once on the common result (early joining). We present a variability-aware interpreter using the white-box strategy in Section 3.

Black-box strategy. The white-box strategy has the disadvantage that we need to modify an existing analysis (usually in a fundamental and crosscutting way, affecting interfaces and internal data structures). Several researchers have investigated how to use existing analyses out of the box instead [2, 23, 27]. They rewrite the product-line implementation or rephrase the specification such that it can be analyzed as a whole with an existing off-the-shelf tool. Typically, we need a powerful existing analysis (such as model checking) that can already deal with some form of variation. Since the analysis tool is reused as is, we name the strategy the black-box strategy.

A typical example of the black-box strategy is to encode an analysis as specification for a model checker. Since model checkers are already capable of dealing with different values of variables, we can encode compile-time variability (as the #ifdef variability from Figure 4 or the Cond and opt elements in our AST) using normal control-flow mechanisms of the host language (as if statements). A model-checking tool then explores all feasible program paths (covering the paths of all configurations). As we encode compile-time variability merely as additional run-time paths, the model checker is able to reason about all configurations. If the model checker detects a violation of the specification, we can reconstruct the erroneous configuration from the problematic execution path. The efficiency of the approach depends on the efficiency of the reused analysis. Modern model checkers already contain sophisticated mechanisms to deal with variations and many paths.

After introducing the basic strategies, let us adapt them for variability-aware testing, first using a white-box strategy (Sec. 3), then with a black-box strategy (Sec. 4).

3. WHITE BOX: A VARIABILITY-AWARE INTERPRETER

As a first attempt to perform variability-aware testing, we implemented an interpreter that is explicitly aware of variability and represents variability locally in its data structures (white-box strategy). For implementing the interpreter, we adopt patterns from prior white-box variability-aware analyses.

A traditional textbook interpreter takes a code fragment, in the form of an AST (without variability), as well as a store; executes the code fragment; and returns an updated store with all variable assignments. In contrast, our variability-aware interpreter takes an AST with variability, a variability context, and a variable store; executes it (covering the entire configuration space); and returns an updated variable store. Let us go through these ingredients one by one:

- **AST with variability.** We execute programs and program fragments given as ASTs with variability, as described in Section 2.1.
- **Variability context.** The variability context (vcts) describes which part of the configuration space we are currently executing. Like presence conditions, we represent the variability context with a propositional formula. For example, \( \text{true} \) means that we are analyzing all configurations, and \( X \lor Y \) means that we are analyzing all configurations in which feature \( X \) or feature \( Y \) is selected. If the variability context is not satisfiable, we do not need to execute that code fragment, because it cannot occur in any configuration. Typically, we aim at executing code within a large variability context (describing many products).
- **Variable store.** Where a traditional store maps names to values \( \text{Map}[\text{String}, \text{Value}] \), a variable store maps names to conditional values \( \text{Map}[\text{String}, \text{Cond}[\text{Value}]] \); so a variable can have different values in different configurations. We store variability as local as possible (cf. Sec. 2.2). If we were dealing with more complicated values, such as objects or functions, we would incorporate variability into the value representation, for example, fields of an object would store conditional values. We show the implementation of our variable store and corresponding access functions in Figure 5 (top).

3.1 Implementation

In Figure 5, we sketch a Scala implementation of our variability-aware interpreter. For illustration, we also show three example traces in Figure 6.

First, the interpreter does not perform any computation if the variability context is not satisfiable, as determined with a SAT solver (Line 10).
Figure 5: Variability-aware interpreter for the WHILE language, encoding variability in all execution steps (excerpt)

When interpreting an assignment (Line 11), we first evaluate the expression to a conditional value in the current variability context, then we store the value. If we execute the statement only in a restricted variability context, we also only store the value in that context.

The case for block statements (Line 14ff) illustrates how we restrict the variability context on optional statements. We execute each statement with a variability context restricted by the presence condition of that statement. If the statement has presence condition true, the variability context remains unchanged.

To evaluate a conditional expression (Lines 37ff), we evaluate every alternative expression separately in the corresponding variability context (Line 39; using auxiliary function condFlatMap defined in Figure 2). Variables are simply looked up in the store (Line 42); negations are applied to all alternative values (Line 44; also using auxiliary function condFlatMap). Notice, how we map over conditional values to preserve potential variability; if the AST does not contain variability, the interpreter behaves like a traditional interpreter.

As a novel concept, we use auxiliary function whenTrue when executing if and while statements (Lines 18–30). First, we evaluate the expression to a conditional value. Now, we need to decide when to execute the body. We want to execute it in all configurations in which the expression’s value is true, but only once. To this end, with whenTrue, we determine a presence condition describing in which configurations the value is true. Subsequently, we execute the body only in the restricted variability context of those configurations in which the expression is true. Note that if the expression’s value is false in all configurations, whenTrue will also return an unsatisfiable variability context false, so the body is never actually executed (Line 10).

Finally, the variability context makes it straightforward to deal with external specifications of valid feature combinations, as typically described in a variability model. We specify valid configurations as a propositional formula and simply pass the formula as the outermost variability context. As a consequence, the algorithm will not execute code related only to invalid feature combinations.

3.2 Discussion

As many existing white-box variability-aware analyses, our interpreter incorporates variability locally in internal data structures (e.g., the store and intermediate values), which facilitates late splitting and early joining (cf. Sec. 2.3).

First, as long as possible, we execute the program with a single variability context, even in conditionals and loops. We split the execution late, only when we actually encounter variability locally in the AST or store. In our example in Figure 4, we execute the first statements only once, even after conditional assignments in Lines 3 and 8, as long as those assigned values are not used. In Figure 6, we see that we never execute any statement of our example twice. In contrast, with a brute-force strategy, we would first generate all products and then execute the initial statements in every product. The local representation of variability ensures that we reason about variability only for variables that actually have different values.

Furthermore, we can join intermediate results (with auxiliary function simplify, not shown). For example, when we assign 0 to a again in Line 13 (Fig. 4), we store only distinct values of a and their corresponding conditions (i.e., we simplify choice(Bar, 0, Choice(FOO, 0, 2)) to Choice(Bar/FOO, 0, 2)). If the variable is assigned to the same value in all configurations, we can join the
intermediate result and store only the single value. Joining can reduce effort in subsequent computations, but executing the join also requires computation effort, so there is a trade-off. However, we leave an empirical evaluation of how relevant joins are in practice for future work.

We have not explored limitations in detail yet. While reflection seems conceptually possible to support (operating on the variable structure of the program), I/O poses a problem. If we cannot provide a variability-aware test environment, we might need to perform testing sequentially from the first occurrence of I/O. The WHILE language does not support I/O; hence, we leave also this problem for future work.

3.3 Experience

We have implemented a variability-aware interpreter for the WHILE language, with additional support for procedures. We can parse WHILE programs with preprocessor directives, like those in Figure 4, using the TypeChef variability-aware parser framework [14]. We are using this implementation to experiment with different strategies (e.g., granularity, different variability representation, when to attempt to join results), and to get a better understanding of which kinds of product-line implementations can be executed quickly and for which the execution resembles the brute-force approach (or is even slower due to the additional SAT solving).

We have developed a generator for random product lines written in the WHILE language and have implemented a testing framework following the pattern outlined in Figure 1. We generate all distinct products from our product line and compare the result of interpreting them without variability to the result of our variability-aware interpreter. Specifically, we do not generate unit tests, but, in a form of differential testing, we simply compare the stores following the equivalence in Figure 1 (4a, 4b). In Figure 7, we show how the variability-aware interpreter improves performance over the brute force approach for 100 generated product lines with at most 6 features (for larger product lines, we were unable to reliably generate random products that terminate, we leave this for future work). Absolute times are within few milliseconds; we gathered times as average from three runs. We can see an overhead for the variability-aware interpreter, but also that it mostly outperforms the brute-force analysis as the product-line size increases.

The implementation, which we currently extend with functions and objects, is available together with the test framework at https://github.com/puschj/Variability-Aware-Interpreter.

4. BLACK BOX: VARIABILITY ENCODING

In addition to implementing a variability-aware interpreter from scratch, we also experimented with performing variability-aware testing with existing tools (black-box strategy). We encoded variability such that we can use an off-the-shelf model checker—JavaPathfinder (JPF) and its extension jpf-bdd [30] in our case—to run test cases for all configurations. We use the model checker to execute the program paths of all valid configurations. This corresponds to separate testing of all configurations in the brute-force approach.

Since model checkers are already capable of dealing with different values of variables, we encode compile-time variability using normal control-flow mechanisms of the host language. For example, we rewrite the code from Figure 4 as shown in Figure 8 (Lines 1–14). We replace preprocessor macros with global Boolean variables (called feature variables; non-deterministically initialized) and #ifdef directives with if statements or conditional expressions. Such rewrites can be performed mechanically; then, we can proceed with an existing analysis on traditional ASTs without variability. In the general case, the encoding can be trickier, but it is always possible to encode alternatives by renaming or code replication at statement level, as explored elsewhere [2, 13, 27]. Even a variability model can be encoded [2, 27]. We call the rewritten product a product-line simulator (a.k.a. meta-product [27]).

After this rewrite, we use JPF to execute test cases. Where the test case on a single product would run deterministically, we introduce nondeterminism through feature variables. Still, JPF explores all feasible program paths of the simulator and gives warnings if one of the paths would result in runtime errors. To illustrate this behavior, we introduced a division-by-zero bug that only occurs when features FOO or BAR are selected (Fig. 8, Line 15). The model checker finds this bug in paths that assign true to FOO or BAR.

Using a model checker for the verification of the simulator is rewarding, because in model checkers “unknown” values for variables are a common concept and model checkers provide out-of-the-box support. However, by using model checking, we limit the set of product lines that can be verified with the approach. For example, we are not able to verify product lines that contain (potentially) endless loops, need user interaction, or need file or network access. For most of these issues, there is advanced research, but we leave those for future work.

4.1 Gray-box extensions: jpf-bdd

Using an off-the-shelf model checker, such as JPF, ensures that errors in all configurations are found. However, in its standard configuration, JPF does not take advantage of the variability information in the product simulator. In the white-box approach, we knew that
variability was always expressed in propositional formulas, and we
could reason about it with SAT solvers and attempt joins. Ideally,
also for model checking, we want an exploration strategy that ex-
ecutes a path until it encounters variability; then it should split the
path, execute both alternatives, and join the paths again as soon as
possible. In the standard configuration, JPF splits paths quite early
(when the variable is assigned to the “unknown” value). Also, stan-
dard JPF never joins paths after variability-related splits, because,
once it has chosen a value for a feature variable, that value is part
of the program state. Because each path has a different choice of
feature values, all paths have at least one difference in their states,
and different states can never be joined. That is, we split late, but
we never join. In the worst case, this results in one execution path
per configuration, much like in the brute-force approach.

Fortunately, JPF is extensible. For product-line verification, we
developed jpf-bdd [30], which enables joining by separating feature
variables from the remaining program state. Feature variables are
stored in separate binary decision diagrams (BDDs). Because the
program states do not contain the feature values any more, JPF can
split paths later and join more states (the extension joins the BDDs
accordingly), so potentially fewer program paths are executed.

In addition, a late splitting optimization in jpf-bdd, which is also
common in other model checkers, chooses the value for feature
variables at the last possible point of (execution) time. In our ex-
ample, this means to store an unknown value for BAR in Line 1
and to choose the concrete value (true or false) only in Line 14.
Lines 4–13 do not depend on BAR, so they only have to be executed
once (for every assignment of FOO). This simple optimization
(late splitting) saves nearly half of the analysis time compared
to a brute-force approach. Still, JPF always splits the entire state,
which corresponds to a store of the form Cond[Map[String,Value]],
and cannot take advantage of sharing between contexts as we do in
our interpreter (using Map[String,Cond[Value]]). Similarly, jpf-bdd
can join stores, but only if they are identical, except for feature
variables.

For more information on jpf-bdd and on performance improve-
ments, we refer to a recent workshop paper [30].

By extending JPF, we diverge from the pure black-box strategy
and actually extend an existing tool. We still reuse most existing
work. Hence, we call this a gray-box strategy. Actually, jpf-bdd
was developed independently of and prior to our testing efforts and is
not specific to product lines. Put differently, we reused the existing
tool jpf-bdd as black-box without further modifications. However,
the fact that the extension was developed by the second author gives
us some perspective on the effort of specific extensions.

4.2 Experience
To gain experience with JPF for variability-aware testing, we rewer-
ted the Graph Product Line [19] as a product-line simulator (as ex-
plained above). The Graph Product Line is a frequently used bench-
mark for product-line technology, a product line with 15 features,
giving rise to 42 configurations, written in about 1000 lines of Java
code, and (slightly) more realistic than the generated Whittle pro-
grams above. We attempted to detect 10 bugs carefully introduced by
Cohen et al. for prior work on testing with sampling strategies [5].
One of the defects introduces an endless loop, so it cannot be found
with JPF. Of the remaining defects, two defects already showed up
with exceptions; for the others, we encoded corresponding specifi-
cations using runtime assertions, analogue to how xUnit unit tests
indicate a failed test with an exception. We executed tests with two
provided test graphs.

We built 10 variants of the product-line simulator (9 variants with
one defect each, and 1 variant without defects). As a baseline, we
tested each of the 42 configurations of each variant in a brute-force
fashion in a standard Java execution environment. Next, we executed
JPF (henceforth called jpf-core) and our extension jpf-bdd on all 10
variants. We report the arithmetic mean of three executions and the
corresponding standard deviation, with 2 GB RAM on two 1 GHz
cores of an Opteron QuadCore machine.

Running tests in the brute-force strategy with Java took 13 ± 0
seconds per product line. In contrast, jpf-core needs 167 ± 50
seconds, jpf-bdd 14 ± 1 seconds per product line.

First, surprisingly, jpf-core is much slower than the brute-force
approach. However the difference can be explained because the
standard Java virtual machine is more optimized than the virtual-
machine part of JPF (which runs a custom byte-code interpreter
written in Java). Executing the brute-force approach with the JPF
virtual machine (deterministic, without performing additional model-
checking overhead) requires 230 ± 7 seconds per product line,
which indicates a conceptual speed-up. As the brute-force approach
behaves exponentially, we expect higher speed-ups in larger product
lines.

Second, jpf-bdd outperforms jpf-core by an order of magnitude,
because it can join many paths. In the Graph Product Line, joins are
particularly effective, because several features have no persistent
influence on the program state. For example, feature Cycle executes
searches for cycles in the graph, prints the result, but does not change
any variables shared with other features; so, jpf-bdd joins where
jpf-core can not.

Though we are at an early stage, our experiment is encouraging
to look at variability-aware testing with (extended) model checkers.

5. RELATED WORK
Product-line testing. As in all other domains, testing has been rec-
ognized as a crucial topic during product-line development. General
strategies, such as those discussed by Pohl et al. [22], emphasize
testing features in isolation (for example, unit tests on plug-ins) and
preparing test cases that should be run on each generated product.
Testing the integration of features remains hard, though. Pohl et al.
distinguish a brute-force strategy from a sampling strategy and an
application-only strategy (only products generated for customers
are tested). They encourage reuse of test artifacts, but they have no
means of testing all configurations of the product line, other than
brute-force.

Along these lines, many researchers have investigated suitable
sampling strategies according to some coverage criteria [5, 8, 18,
20, 21, 24]. A typical strategy is sampling with n-way feature cover-
age, such that each n-tuple of features appears in at least one tested
product [20]. Especially, 2-way feature coverage is frequently used,
since it seems to strike a good balance between number of products
that need to be tested and detection of interaction problems [16].
Nonetheless, sampling prevents establishing properties about the
entire product line.

Another strategy to scale product-line testing is to determine
which test cases need to be run in which configurations, to reduce
the number of test executions. Kim et al. have used static analysis
to conservatively approximate which test cases are influenced by
which features [15]. Shi et al. have used symbolic execution to an-
alyze the product line to reduce the number of products that need
to be tested [24]. Cichos et al. explore a strategy to generate tests
to achieve coverage for an entire product line [8], and Lochau et
al. use test case generation such that products can be tested
incrementally [18]. All these approaches analyze the whole product
line (or its test model) in a variability-aware fashion to reduce the
number of tests, but the tests themselves are still executed on indi-
vidual products. In contrast, by construction, our interpreter and our
encoding with model checking cover the entire product line and split test execution only when needed, without dedicated prior analysis.

Variability-aware analysis. Although a rather recent research topic, many researchers have investigated strategies for variability-aware analysis for parsing (white-box [14]), type checking (white-box [1, 7, 13, 25] and black-box [23]), model checking (white-box [9, 17] and black-box [2, 23]), static analysis (white-box [4] and black-box [3]), and theorem proving (black-box [27]). For a detailed overview of that field, we defer the interested reader to a recent survey [26].

The specific style of writing a variability-aware analysis by mapping over conditional data structures was inspired by variational programming by Erwig and Walkingshaw [11, 12]. They also presented and formalized a type system for the lambda calculus in this style [7]. Our encoding differs from theirs in that we encode choices and feature models with arbitrary propositional formulas, instead of using atomic feature names defined within the conditional data structure. This difference makes our approach potentially simpler and more flexible, but also more expensive to compute (we rely on SAT solvers or BDDs).

Our interpreter implements a form of mixed concrete/symbolic execution—see [6] for an overview of that field. Conceptually, in the variability-encoded version, we consider all feature variables as symbolic and execute the remaining program with concrete values. We have not yet experimented with existing tools for symbolic execution. They seem promising as black-box tools for the variability-encoding strategy. There is a rich and advanced collection of tools to explore for product-line testing in future work.

6. DISCUSSION AND CONCLUSIONS

We have investigated variability-aware testing with a white-box strategy (variability-aware interpreter), a black-box strategy (variability encoding for JPF), and even a gray-box strategy (variability encoding for jpf-bdd). In all cases, we run a test case on all configurations of a product line at once, as opposed to a brute-force or sampling strategy. Although it is too early to draw sound conclusions, we want to share our observations and encourage feedback at this early stage. We have gained interesting insights into the spectrum between white-box, gray-box, and black-box analyses regarding implementation effort and flexibility.

Effort. The white-box strategy obviously requires more effort to implement than the black-box strategy. We need to write our own interpreter from scratch or significantly rewrite an existing interpreter, because variability pervades all data structures and execution steps. While writing interpreters is well understood, writing an interpreter for a full language such as Java, C, or JavaScript requires significant effort. In contrast, reusing existing and optimized tools in the black-box strategy allowed us to experiment directly with Java code with much less effort.

Flexibility. The white-box strategy is more flexible than the black-box strategy. The black-box strategy depends very much on the power of the existing analysis and how efficiently it deals with variability. We have to 'hope' that their optimizations fit to our use cases (test case execution despite variability in our case). The variability encoding does not necessarily have the shape of typical programs for which general-purpose analysis may be optimized.

Product-line analysis is special in that variability follows only few restricted patterns, reducible to propositional formulas and Boolean satisfiability problems. Those specifics are usually not considered by the black-box tools or might even get lost in the encoding (i.e., analyzing arbitrary expressions in if statements is much harder than analyzing presence conditions in choice nodes). By extending existing tools (gray-box strategy: jpf-bdd in our case), we can attempt to add some product-line specific optimizations. In the white-box strategy, however, we have full control over the execution and how to store variability internally. We can weigh where and how to encode variability (e.g., Cond[Map[T, U]] vs. Map[T, Cond[U]]), when to join results, and so forth. We exploit that variability is always expressed with propositional formulas, allowing more specific analyses, such as the one we performed with whenTrue.

We illustrate the difference in internal behavior between our variability-aware interpreter and the strategy of JPF with a constructed favorable example of the factorial function in Figure 9. The interpreter attempts to execute the body of the while loop three times. The first time with variability context true, that is, all values are updated together. Only in the second iteration, the body is executed in a restricted context; so, all values are updated conditionally. The final iteration then has variability context false and is not executed at all. This is an instance of storing variability locally and splitting as late as possible. In the same example, JPF (and jpf-bdd) separately computes the while loop 1 and 2 times without any sharing. This constructed example can demonstrate significant performance differences between both strategies, when using larger values for a.

We are still exploring different strategies within the spectrum between pure white-box and pure black-box approaches. The gray-box strategy appears promising, although extending existing black-box tools depends on predefined interfaces. Also experimenting further with white-box implementations should yield useful insights in the specifics of product-line testing. As next step, we want to grow our interpreter to support a real language. We are still at the beginning of the road to variability-aware testing and encourage others to join this path.

Acknowledgments. This work is supported by ERC grant ScalPL #203099 and the DFG grants AP 206/2, AP 206/4, and LE 912/13. We thank Myra Cohen for sharing the GPL bug scenarios.

7. REFERENCES

Conditioned Model Slicing of Feature-Annotated State Machines

Jochen Kamischke  
Institut für Programmierung und Reaktive Systeme  
Mühlenpfordtstr. 23  
Braunschweig, Germany  
j.kamischke@tu-bs.de

Malte Lochau  
Institut für Programmierung und Reaktive Systeme  
Mühlenpfordtstr. 23  
Braunschweig, Germany  
lochau@ips.cs.tu-bs.de

Hauke Baller  
Institut für Programmierung und Reaktive Systeme  
Mühlenpfordtstr. 23  
Braunschweig, Germany  
h.baller@tu-bs.de

ABSTRACT

Model-based behavioral specifications build the basis for comprehensive quality assurance techniques for complex software systems such as model checking and model-based testing. Various attempts exist to adopt those approaches to variant-rich applications as apparent in software product line engineering to efficiently analyze families of similar software systems. Therefore, models are usually enriched with capabilities to explicitly specify variable parts by means of annotations denoting selection conditions over feature parameters. However, a major drawback of model-based engineering is still its lack of scalability. Model slicing provides a promising technique to reduce models to only those objects being relevant for a certain criterion under consideration such as a particular test goal. Here, we present an approach for slicing feature-annotated state machine models. To support feature-oriented slicing on those models, our framework combines principles of variability encoding and conditioned slicing. We also present an implementation and provide experimental results concerning the efficiency of the slicing algorithm.

Categories and Subject Descriptors
D.2.4 [Software Engineering]: Software/Program Verification; D.2.13 [Software Engineering]: Reusable Software, Reuse Models

General Terms
Design, Theory

Keywords
Software Product Lines, Model-Based Software Engineering.

1. INTRODUCTION

Model-based software engineering provides a rich collection of modeling languages and corresponding techniques for the specification, documentation, maintenance, and verification/validation of high-quality software systems in a systematic way. In particular, modeling approaches for specifying the operational behavior of a software system are often based on state-transition diagrams such as UML state machines [18]. Thereupon, various applications of those models to verification/validation techniques like model-based testing and model checking have been proposed [5, 1].

However, the major drawback of those approaches is still their lack of scalability. This is even worse in the presence of explicit variability at model level as apparent, e.g., in software product line engineering [15]. Herein, models are enriched with capabilities to specify common and variable parts occurring in a family of similar product variants. For instance, model elements are annotated with selection conditions, i.e., propositional formulas over feature parameters to guide the assembling of model variants w.r.t. a particular feature configuration [5]. Hence, such feature-annotated models integrate any potential behaviors of all product variants of an SPL within a virtual, so-called 150% model. This additional model dimension as tailored by the valid product configuration space of an underlying domain feature model [11] further complicates the application of model-based analysis techniques to variant-rich, real world problems represented by an SPL.

Model slicing provides a promising approach to handle the complexity problem of behavioral models by performing static, i.e., syntactical model reductions that preserve some aspects of model semantics w.r.t. a slicing criterion under consideration [22, 21]. Therefore, model slicing extracts those model parts affecting certain computational units only and, at the same time, ensuring the resulting model slice to preserve a syntactically well-formed model structure. Slicing has gained applications in various fields of program analysis for reverse engineering, program integration, software metrics, component reuse, etc. [3]. Model slicing adopts the concepts of program slicing to reduce verification/validation efforts. For instance, in model-based testing, choosing test goals as slicing criteria allows for efficient test case generation, debugging, and change impact analysis during regression testing, whereas slicing along a certain model property, e.g., given as an LTL formula, decreases model checking complexity. However, recent model slicing approaches are incapable to cope with models enriched with feature annotations. In the presence of variable model parts, the further model dimension has to be taken into account when slicing for a particular criterion in order to yield a well-formed
model for every model variant that contains all parts relevant for the criterion.

In this paper, we present a framework that enhances model slicing to state machine models with explicit variability in terms of feature annotations. Besides well-known slicing-based model analysis techniques, we use conditioned slicing [19] to enrich slicing criteria with constraints over feature values thus constituting a (partial) product configuration the slice is to be derived from. Therefore, we use variability encoding [5] to embed feature annotations into models such that slicing algorithms are able to treat them as regular computational units. This allows the analysis of behavioral commonality and variability among product variants, e.g., for efficient verification of complete product families [4, 1], feature interaction detection [20, 12] and incremental SPL testing [13]. The concepts are illustrated by a running example based on the Vending Machine SPL [4, 8] and evaluated by means of a case study from the automotive domain [14]. We also present a sample implementation and evaluation results for our approach.

The paper is organized as follows. In Sect. 2, we review basic notions of state machines and how to enrich them with explicit feature annotations to model variable behavior in the SPL context. In Sect. 3, we outline a model slicing algorithm for state machines and extend it to be applicable to feature-annotated state machines. In Sect. 4, we present an implementation based on variability encoding and conditioned slicing and show results of some experiments performed. Sect. 5 concludes.

2. STATE MACHINE MODELS WITH FEATURE ANNOTATIONS

We first review the common modeling concepts of state machines. For a detailed survey of the abstract syntax and formal semantics of various state machine variants, we refer, e.g., to [7]. We then describe extensions to model behavioral variability among different SPL product variants using feature-annotated state machines with explicit selection conditions over feature parameters organized in a domain feature model.

2.1 State Machine Models

State machines provide behavioral specifications of (software) systems by means of computational states \( s \in S \) and transitions \( t = (s, l, s') \in T \) leading from a source state \( s \in S \) to a target state \( s' \in S \), where label \( l \) denotes (re-)actions of the system. Originating from Harel’s STATECHARTS [10], various variants and implementations of state-machine-like modeling approaches appeared, e.g., UML state machines [18] and MATLAB/SIMULINK/STATEFLOW [6].

State machines are represented as state-transition graphs enriched with several extensions for modeling complex system behavior. Vertices are visualized as rectangles with curved edges and denote states. Transitions between states are visualized as directed, labeled edges. A sample state machine model for the control logics of the Extended Vending Machine (EVM) SPL case study is depicted in Fig. 1. The Vending Machine SPL originates from a case study presented in [4, 8]. The state machine is divided into three concurrent parts. The left part is the vending machine itself with functionality for producing sugared and non-sugared coffee and cappuccino. In the middle, we extend the original machine by a milk administration system, that tracks the milk usage an produces a warning if a predefined amount of milk is consumed. The right part gives an advise to administration staff to refill milk. The control cycle of the vending machine lets the user first insert money and then to choose whether he/she prefers sugared or non-sugared beverage. In the second step, the user has to choose the desired beverage which is then produced. When finished, a ring tone is played.

Labels \( l \) of transitions \( t = (s, l, s') \) specify the visible, event-based behavior of the system. We sometimes write \( s \to s' \) for short, where \( l \) might be omitted if not relevant. Labels consist of two parts, a trigger and an action. The transition trigger denotes an input event \( e_t \) to occur for releasing the transition, whereas the transition action denotes an output event \( e_o \) emitted as system reaction to this input.

The set of behaviors specified by a state machine is given as the set of, potentially infinite, sequences of action/react pairs that correspond to valid paths, i.e., a consecutive sequence of transitions in the state-transition graph starting from a well-defined initial state \( s_0 \). In Fig. 1 a sample sequence of action/react pairs is

\[
(1 \in \text{cup} \text{ act} \text{ ring_u_tone}), (\text{cup}_\text{act} \text{ ring_u_tone}), (\text{cup}_\text{act} \text{ ring_u_tone})
\]

where the corresponding valid path is

\[
s_0 \xrightarrow{1} s_1 \xrightarrow{} s_3 \xrightarrow{} s_9 \xrightarrow{} s_{11}
\]

A transition with an empty trigger part is always enabled whenever its source state is active. Branches in the state-transition graph specify alternative behavior, e.g., the choice between sugar and no_sugar in state \( s_1 \) yields two different subsequent paths. Loops specify reactive behavior, e.g., the EVM returns to the initial state \( s_0 \) after a service is completed. State machine labels are often further extended to complex transition labels, e.g., incorporating conditional guarding expressions \( G \) in addition to the triggering event and computational statements in the action part, both accessing and/or changing values of internal variables \( v \) to specify internal data flows. In the EVM example, an integer variable \( \text{Milk} \) is used to store the current amount of milk. We use the following notation for transitions labels

\[
e_t [ G ] / \{ \text{act} \}_c_o
\]

where computational actions in the \( \{ \text{act} \} \) component are given as assignment statements on internal variables in our example. Again, all label components are optional. Complex transition labels cause computational states of a state machine under execution to be further enriched by internal status information, e.g., comprising variable values.

Considering the state structure, recent state machine modeling approaches provide hierarchical, as well as parallel decomposition of states by means of nested sub machines. Hierarchical state decomposition defines a sub state relation \( \preceq \subseteq S \times S \) thus for nesting state machines into states. The state \( s_{17} \) in our example is extended by adding two sub states \( s_{18} \) and \( s_{19} \). If \( s_{17} \) is entered, \( s_{18} \) is simultaneous entered as it is marked as initial state of this sub machine. Two states \( s, s' \in S \) not being related under \( \preceq \) are excluding each other, denoted by a state exclusion relation \( \not\subseteq \subseteq S \times S \) (for example states \( s_{16} \) and \( s_{18} \)). In state machines with concurrent
state decomposition, two states $s, s' \in S$ not related under $\prec$ might be also related under the orthogonal state relation $\perp \subseteq S \times S$. Concurrent sub machines are graphically divided by dashed lines. If a sub machine is active during an execution, all the concurrent sub machines are also active. In the EVM, we have, e.g., $s_0 \perp s_{15}$.

For a state-transition diagram to obey well-defined operational semantics, further well-formedness properties are to be satisfied.

**Well-formed State Machines.**

Depending on the syntactical constructs and related semantics provided by the different state machine modeling approaches, the corresponding well-formedness criteria may differ, accordingly. We consider the following exemplary constraints.

1. $(S, \prec)$ forms a finite rooted tree on the set $S$ of states,
2. for the source state $s$ and target state $s'$ of transitions $(s, t, s') \in T$, it holds that $s \nless s'$,
3. the transition graph of each sub machine is connected, i.e., for each state $s_k \in S$, there exist a path $t_1 t_2 \cdots t_k$ such that $t_k = (s_{k-1}, l, s_k)$ and $t_1 = (s_0, l, s_1)$, where $s_0$ is the initial state of the corresponding sub machine and every state $s_i$, $0 \leq i \leq k$, has the same parent state w.r.t. $\prec$.

Further well-formedness criteria, e.g., concerning the compatibility of input/output event alphabets of different sub machines are out of scope in the following (cf. [7] for details).

### 2.2 Feature-Annnotated State Machines

Software product line engineering (SPLE) propagates the exhaustive reuse of design artifacts between similar product variants throughout all development phases [15]. SPLE is based on a generic platform for assembling implementations of different product variants. This instantiation is determined by the features selected in the product configuration. Besides assemblies of final product implementations, this principle is also applied in earlier stages, e.g., by means of reusable behavioral models for variable behavioral abstractions for the different product configurations. Various approaches for enriching state-machine-like models with feature-oriented variability capabilities appeared in the literature. In annotative approaches, model elements obey selection conditions in terms of propositional formulas over features. A particular model variant is then derived from such a 150% model by projecting those elements whose selection conditions are satisfied by the corresponding product configuration. For instance, in the FTS approach of Classen et al., transitions of a labeled transition system are annotated [5], whereas in [6], STATEFLOW models are used, and in [9], UML state machines are considered. In compositional approaches, model variants are assembled by combining feature-specific model artifacts according to the product configuration chosen [17]. In transformative approaches, model variants are obtained from an arbitrary core model by applying sets of delta operations to that core whose application conditions are satisfied by the product configuration [16, 13]. Besides those explicit couplings of variable model elements to feature parameters, implicit approaches use, e.g., modal transition systems to distinguish between mandatory transitions of the core model and optional parts. Thereupon, Asirelli et al. use deontic logics over feature parameters to further constrain artifact assemblies [1].

In our slicing framework, we use explicit annotations of state machine elements with selection conditions over feature parameters organized in a feature model. In general, we assume an SPL to define a finite set $F = \{f_1, f_2, \ldots, f_n\}$ of (boolean) feature parameters denoting the main increments of (variable) functionality of the different product variants. The product space of an SPL is defined by the set of all potential product configurations $\Gamma$ over those features. Here, we assume a product configuration to be given as a mapping $\Gamma : F \rightarrow \mathbb{B}$ assigning a boolean value from $\mathbb{B} = \{false, true\}$ to features $f \in F$, where $\Gamma(f) = true$ states feature $f$ to be selected, whereas $\Gamma(f) = false$ states feature $f$ to be unselected in the respective configuration. A configuration $\Gamma$ is a partial configuration if $\Gamma$ is a partial function on $F$, and it is a full configuration, otherwise.

Feature models restrict product spaces to valid product spaces of SPLs by imposing additional constraints on feature combinations. A common graphical representation of feature models, initially proposed by Kang et al. in [11], or-

![Figure 1: State Machine Model of an Extended Vending Machine Product Variant.](image-url)
organizes features in a tree-like hierarchy. The sample FODA feature diagram for the EVM SPL is shown in Fig 2. The tree hierarchy imposes a decomposition of a parent feature into sets of child features such that a selection of a child feature requires the selection of its parent feature in a product configuration. Sibling child features can be grouped, where the group type constraints the possible combinations of those features. For instance, an alternative group consists of the features cur and usd below feature cur. Hence, one and only one of these features must be selected into a valid configuration. In contrast, the features co, t and ca are organized in an or-group and can therefore be arbitrarily combined if at least one of them is selected. Finally, feature diagrams provide cross tree constraints among features given as require edges, e.g., from ca to co, and exclude edges.

According to [2], we assume a feature model $FM \in \mathbb{B}(F)$ to be given as a propositional formula over feature parameters. Hence, the valid product space is given as

$$PCFM = \left\{ \Gamma : F \to \mathbb{B} \mid \Gamma \models FM \right\}$$

thus containing only those (partial and full) product configurations satisfying $FM$. We use the set $F$ of feature parameters to annotate variable state machine elements with feature-oriented selection conditions. A feature-annotated state machine model for a feature model $FM \in \mathbb{B}(F)$ is defined via an annotation function

$$\alpha : \mathcal{E} \to \mathbb{B}(F)$$

that assigns to syntactical modeling entities $e \in \mathcal{E}$ a selection condition $\alpha(e) \in \mathbb{B}(F)$ by means of a propositional formula over feature parameters in $F$. By convention, we require $\alpha(e) \models FM$, i.e., the selection condition satisfies the constraints of the feature model. Considering state machines, the set $\mathcal{E}$ of syntactical elements contains, e.g., the set of states, transitions, etc. Element $e \in \mathcal{E}$ of a feature-annotated state machine model is selected into the state machine variant for a configuration $\Gamma \in P_{CFM}$ if $\Gamma \models \alpha(e)$ holds, i.e., the feature parameterization in $\Gamma$ satisfies the selection condition of $e$. For mandatory elements $e \in \mathcal{E}$, where $\Gamma \models \alpha(e)$ holds for any $\Gamma \in P_{CFM}$, we omit the annotation.

The feature-annotated state machine model for the EVM SPL is shown in Fig. 3. By adding annotations to states and transitions, the behavior of the state machine is parameterized by feature selections. For example, the transition $s_3 \to s_9$ is annotated with co, i.e., the corresponding behavior is only relevant for the feature coffee. The derivation of the state machine variant as shown in Fig. 1 results for a configuration $\Gamma$ with

$$v \land bev \land cur \land co \land \neg t \land ca \land r \land \neg usd \land cur$$

thus removing all elements whose selection condition does not satisfy this term.

**Well-formed Feature-Annotated State Machines.**

A feature-annotated state machine implicitly defines a family of state machine variants, one for each valid product configuration $\Gamma \in P_{CFM}$. Hence, the notion of well-formed state machines can be naturally extended to feature-annotated state machines by requiring every derivable state machine variant to be well-formed. Constructive criteria for ensuring well-formed feature-annotated state machines are given as follows.

1. For each state $s \in S$ with $s' \prec s$, it holds that $\alpha(s) \Rightarrow \alpha(s')$, i.e., the presence of a state inductively ensures the presence of all parent states up to the root state,
2. for each transition $t = (s, l, s') \in T$, it holds that $\alpha(t) \Rightarrow \alpha(s)$ and $\alpha(t) \Rightarrow \alpha(s')$, i.e., the presence of a transition ensures the presence of its source and target state, and
3. each state $s_k \in S$ is reachable via at least one path $t_1 t_2 \cdots t_k$ such that $\alpha(s) \Rightarrow \alpha(t_i)$ for $1 \leq i \leq k$.

These requirements may be weakened such that only the set of state machine variants for full product configurations must be well-formed.

3. **FEATURE-ORIENTED SLICING OF STATE MACHINES WITH FEATURE ANNOTATIONS**

We now present our conditioned slicing framework for variable state machine models. We first review the fundamental concepts of state machine model slicing and then enrich the approach to feature-annotated state machines.

**3.1 Model Slicing and State Machine Slicing**

Initially introduced by Weiser, slicing imposes a static, i.e., syntax-based, order preserving projection on program statements yielding a well-formed, reduced program that preserves program semantics w.r.t. a slicing criterion [22]. For instance, a static slicing criterion $(P, V, n)$ projects from a program $P$ those statements into a reduced program $P'$ that affect, i.e., the values of a subset $V \subseteq V_P$ of the program variables $V_P$ of $P$ at program point $n$. In a conditioned slicing criterion $(P, V, n, \Phi)$ an additional condition $\Phi \in \mathbb{B}(V_P)$ over values of program variables in $P$ is given thus reducing the set of potential initial program states for $P$ to be preserved in $P'$ to only those satisfying $\Phi$. Finally, a dynamic slicing criterion contains an explicit initial state thus leading to a slice for a single program execution. Static program slicing algorithms perform reachability analysis by traversing the program dependencies graph and concerning control and/or data dependencies among syntactical program objects affecting the criterion. This slicing is called backward-slicing. The effects the criterion has on subsequent
elements is considered via forward slicing (see [19, 3] for further reading). Generalizing the concept of program slicing to arbitrary behavioral specifications requires the adaption of the corresponding notions for slicing criteria, dependencies and semantics preserving reductions to the respective definitions for slicing criteria, dependencies among related state machine elements, namely sequential, conflicting and hierarchical control dependencies among state and/or transitions, as well as data dependencies due to concurrent accesses to shared variables and synchronization via internal event broadcasts.

- **Parallel Dependency** $\rightarrow_{pd}$ between concurrent elements w.r.t. $\perp$, e.g., $s_0 \rightarrow_1 e_2 \rightarrow s_1$ and $s_{15}$

- **Sequential Data Dependency** $\rightarrow_{sdd}$ between elements being sequentially ordered and accessing the same variable, e.g., $s_{15} \rightarrow s_{16}$ and $s_{16} \rightarrow s_{15}$.

- **Parallel Data Dependency** $\rightarrow_{pdd}$ between concurrent elements w.r.t. $\perp$ accessing the same variable, e.g., $s_{15} \rightarrow s_{16}$ and $s_{21} \rightarrow s_{20}$.

- **Synchronization Dependency** $\rightarrow_{sd}$ between concurrent elements w.r.t. $\perp$, where one generates an event that the other consumes, e.g., $s_{10} \rightarrow s_{11}$ and $s_{15} \rightarrow s_{16}$.

- **Transition Control Dependency** $\rightarrow_{tcd}$ between transitions being sequentially ordered and where one generates an event the other consumes (not here).

- **Global Control Dependency** $\rightarrow_{gcd}$ between states and transitions, where the state is the source of the transition and the transition is triggered by an input event, e.g., $s_1 \rightarrow s_3$ and $s_1$.

- **Refinement Control Dependency** $\rightarrow_{rcd}$ between a state and the initial states of all its sub states w.r.t. $\prec$, e.g., $s_{17}$ and $s_{18}$.

The state machine slicing algorithm (Algorithm 1) is based on these dependencies such that a slice for an element $e$ contains at least all those elements $e'$ on which $e$ depends. Hence, when applied to a well-formed state machine model

Algorithm 1 State Machine Slicing Algorithm.

1: **input**: State Machine $M$, Slicing Criterion $C = e \in \mathcal{E}$
2: **output**: Slice $M_C$
3: $\text{Dep}_M := \text{computeDep}(M)$;
4: $M_0 := \text{initSlice}(M, C)$;
5: **repeat**
6: $M_{i+1} := \text{reachable}(M'_i, \text{Dep}_M)$;
7: $M'_{i+1} := \text{wellformed}^{-1}(M_{i+1})$;
8: **until** $M'_{i+1} = M'_i$
9: $M_C := \text{wellformed}^{-1}(M'_{i+1})$;
the algorithm first calculates $\Delta \mathcal{P}_M$ (line 3). The initial slice $M_0$ created in line 4 solely contains the model element $e$ of the slicing criterion $C$. The final slice $M_C$ is given as the least fix point (1) that contains all model elements approximately having behavioral influences in executions involving $e$, and (2) that is well-formed. Therefore, the iteration in lines 5–8 incrementally extends intermediate slices $M_i$ to $M_{i+1}$ by traversing further model elements reachable from $M_i$ via dependencies in $\Delta \mathcal{P}_M$. In the post-processing step of line 7, the resulting slice $M_{i+1}$ is potentially further enriched with model elements by the routine marked with (+) to obtain slice $M'_{i+1}$ satisfying the well-formedness properties 1 and 2 in Sect. 2.2. The fix point is reached if no further elements are added to the current slice (line 8). In general, the resulting slice $M'_{i+1}$ potentially contains dispensable model elements, e.g., states although being contained in the transitive closure of $\Delta \mathcal{P}_M$, however not being reachable in the sliced state-transition graph. Such elements are removed by the routine marked with (-) in the last step in line 9 to also satisfy well-formedness property 3 in Sect. 2.2.

Considering the complexity of slicing a state machine $M$ built from a set $\mathcal{E}$ of model elements, we have $O(|\mathcal{E}|^3)$ for the computation of $\Delta \mathcal{P}_M$ in line 3, the fix point calculation (line 5–8) lies in $O(|\mathcal{E}|^2)$ and for the reductions in line 9, we have $O(|\mathcal{E}|^3)$. Hence, the overall complexity lies in $O(|\mathcal{E}|^3)$.

We now extend the slicing algorithm to handle feature-annotated state machines.

### 3.2 Extending Model Slicing to Feature-Annnotated State Machines

We first give intuitive examples how to enhance the slicing approach such that it is also applicable to feature-annotated state machine models. Therefore, we discuss sample slicing scenarios on the feature-annotated state machine model introduced in Fig. 3. According to the extension of state machines via selection conditions $\alpha(e) \in \mathcal{B}(F)$ over features in $F$ for model elements $e \in \mathcal{E}$, slicing criteria for such feature-annotated state machine models are likewise extended to now contain (1) a model element $e$ of the feature-annotated state machine model as usual, and (2) a (partial) product configuration $\Gamma \in PC_{PM}$ denoting the state machine projection for which the slice for $e$ is to be constructed.

As a first example, we consider $\Gamma = \{ca \rightarrow false\}$, i.e., a partial configuration in which feature $ca$ is unselected, whereas element $e$ is left open. The resulting state machine slice is shown in Fig. 5. Thus, the transitions $s_2 \rightarrow s_6$ and $s_3 \rightarrow s_7$ are removed as both are annotated with this feature. Furthermore, state $s_6$ is removed, because its only incoming transition is $s_2 \rightarrow s_6$, thus also causing the transition $s_6 \rightarrow s_7$ to be removed. This further implies the consequitively removal of the elements $s_7$, $s_7 \rightarrow s_{10}$, and $s_{10} \rightarrow s_{11}$. After removing $s_{10} \rightarrow s_{11}$, there is no transition left in the intermediate slice that potentially triggers action pour_milk. Therefore, the transition $s_{15} \rightarrow s_{16}$ can not be released any more and is also removed. As a consequence, $s_{16} \rightarrow s_{15}$, $s_{15} \rightarrow s_{17}$, and $s_{17}$ with all its nested states and transitions are removed from the slice. With the removal of $s_{15} \rightarrow s_{15}$, the internal event eventMilkError is not released by any transition anymore. Thus, $s_0 \rightarrow s_{14}$ and $s_{14}$ are also removed. Furthermore, the sub machine containing only the initial state $s_{13}$ is removed from the model, as it has no behavior left. Finally, the transition $s_{21} \rightarrow s_{20}$ is removed, because the internal variable Milk is no longer written in any transition. When further assuming the model element $e$ to be added to the slicing criterion in terms of state $s_{20}$, then state $s_{21}$ and transition $s_{20} \rightarrow s_{21}$ are also removed.

As a second example, assume the configuration $\Gamma$ in the criterion to not contain the feature $co$. As a first step, the transitions $s_2 \rightarrow s_4$ and $s_3 \rightarrow s_9$ are removed, thus causing the state $s_9$ to be removed after removing the elements $s_4$ and $s_4 \rightarrow s_9$. Thereupon, transition $s_9 \rightarrow s_{11}$ is removed. Due to the require edge in the feature model in Fig. 2 between the features $ca$ and $co$, each configuration without $co$ must not include $ca$. Thus, the transitions $s_2 \rightarrow s_6$ and $s_3 \rightarrow s_7$ are also removed. This leads to similar additional removals of further states and transitions as in the first example.

### 3.3 Feature-Oriented State Machine Slicing

To provide a feature-oriented model slicing framework that deals with feature-annotated state machine models as described above, we enhance our approach correspondingly. Therefore, Algorithm 1 is extended to conditionally slice for (partial) product configurations w.r.t. the feature annotations. Hence, the input of the algorithm is extended to a well-formed feature-annotated state machine $(M, \alpha)$ and a feature-oriented slicing criterion $(e, \Gamma)$ with $\Gamma \in PC_{PM}$. The slice extension in line 6 is adapted such that if $M'_i$ already contains model element $e \in \mathcal{E}$, whereas element $e'$ is not yet part of $M'_i$ and there is a dependency $\rightarrow_{\Delta \mathcal{P}_M}$ such that $c \rightarrow_{\Delta \mathcal{P}_M} c'$, then $c'$ is added to $M_{i+1}$ only if $\Gamma \models \alpha(e')$ holds, i.e., the selection condition satisfies $\Gamma$. In contrast, no adoptions are required for the step performed in line 7. As we require the feature-annotated state machine $(M, \alpha)$ to be well-formed, the extension of intermediate slices $M_{i+1}$ to well-formed slices $M'_{i+1}$ is always possible by solely adding further elements that also satisfy $\Gamma$. The removal of dispensable elements in line 9 is done as before, as for removals of unreachable elements from the slice, the corresponding annotations can be ignored.

Considering the complexity of the enhanced slicing algorithm, the SAT check introduced in line 6 naturally implies NP-completeness. In Sect. 4, we investigate complexity issues for the approach by considering slicing scenarios on a sample case study.
4. IMPLEMENTATION, EXPERIMENTS AND EVALUATION

We now describe an implementation based on variability encoding and conditioned model slicing and present a sample tool chain for our slicing framework. On this basis, we performed experiments considering the efficiency of the algorithm.

4.1 Implementation and Tool Chain

Our implementation is based on the following techniques.

- **Variability Encoding:** We integrate the feature annotations into state machine model semantics by (1) adding a fresh boolean variable $v_f$ for each feature $f \in F$, and (2) embedding the selection conditions over those variables into transition guards.

- **Conditioned Slicing:** The product configuration specification component $\Gamma \in \mathbb{B}(F)$ of a slicing criterion $(e, \Gamma)$ defines a conditioned slicing criterion $(e, \Phi)$ on the variability encoded model such that the initial condition $\Phi = \Gamma$ must hold for the feature variables introduced for the transitions traversed in the slice.

The variability encoding approach originates from the work of Classen et al. [5]. A schematic illustration of variability encoding is shown in Fig. 6. The example shows the variability encoding of a state and a transition. The state is annotated with the feature term $f_1 \& \& f_2$ as shown in (a). After variability encoding, the guards of the two incoming transitions are extended by a conjunction of that term and the existing guard. As the guards of the transitions were initially empty, the conjunction $true \& \& (f_1 \& \& f_2)$ is reduced to $f_1 \& \& f_2$. The annotation $f_1 \& \& \neg f_2$ of the transition $s_1 \rightarrow s_2$ is embedded into the transition guard thus disabling this transition, e.g., feature $f_1$ is deselected and feature $f_2$ is selected in a configuration by initializing $f_1$ with the value $false$ and feature $f_2$ with the value $true$. For every sub machine located below a state, the algorithm is executed recursively propagating feature guards of transitions leading to this state also to all its sub machines.

Based on this concept, we developed a tool chain for conditioned slicing of variability encoded state machine models as shown in Fig. 7. We use IBM RATIONAL RHAPSODY as graphical front end for modeling full-fledged state machines and import them into our slicing framework via a COM interface. As a second input, a feature model as described in Sect. 2.2 is considered. For feature modeling, we use PURE::VARIANTS developed by pure-systems. In addition to a graphical user interface, PURE::VARIANTS offers a RHAPSODY plug-in that allows the user to annotate the state machine model with selection conditions over features in the feature model. A conditioned slicing criterion is to be provided by the user consisting of a state machine element $e$ and a propositional formula $\Phi$ over feature names. The slicing algorithm is implemented in Java and works in several stages as described above. For the satisfiability checks, we apply the SAT-Solver SAT4J. The resulting slice is re-imported into RHAPSODY.

4.2 Experiments and Evaluation

We evaluated the slicing algorithm w.r.t. efficiency, i.e. the amount of model reductions achievable in sample slicing scenarios. We applied the implementation to an automotive case study, a simplified Body Comfort System (BCS) SPL, including features such as an alarm system, power window control, etc. (cf. [14] for details). For our experimental evaluation, a representative set of 7 product variants with an ascending number of features is used. Fig. 8 shows the result of dependency and slice calculation. The first two diagrams contain the number of dependencies in the different increments of the slicing process. We differentiate between parallel dependencies, which are not used for slice calculation but for further dependency calculation, and dependencies used for slice calculation. The three states in which the number of dependencies are compared are before the slicing (horizontal line), after the application of the criterion to the dependencies (the line in light grey) and after slicing. A high number of parallel sub machines induces a corresponding amount of concurrent dependencies. The slicing and especially the removing of all unreachable parts in the state machine leads to the difference between the second and the third stage. The third diagram shows the number of elements in the feature-annotated state machine before slicing and in the product variant models after slicing. Most feature annotations are rather simply structured and solely refer to either one, or at most three features. Thus the conditioned slicing scales well even for models with a high number of model elements. Since the BCS case study has few annotations with negated feature variables, the number of elements in the slice is increasing with a growing number of features. But in general there is no direct correspondence between the number of features and the number of elements in a slice.

As an application example, we applied the framework to
Considering two features applicable to support feature interaction detection [12, 14]. Furthermore, the conditioned slicing is change impact analysis for model-based testing of software product lines [13]. Additionally, the approach integrates principles of variability encoding and conditioned slicing. As a future work, we plan to integrate the implementation into a model-based SPL testing framework in order to perform change impact analysis among different product variants for automated derivation of retesting obligations [13]. The approach also paves the way to semi-automatic feature interaction detection. Furthermore, an application of the concept also to other kinds of models as well as feature-oriented program slicing seems promising. In addition, we plan further experiments and improvements concerning the accuracy and efficiency of the approach.

5. CONCLUSION

We presented a conceptual framework and a sample tool chain for a feature-oriented slicing approach of variability enriched state machines models. The approach adopts principles of variability encoding and conditioned slicing. As a future work, we plan to integrate the implementation into a model-based SPL testing framework in order to perform change impact analysis among different product variants for automated derivation of retesting obligations [13]. The framework also paves the way to semi-automatic feature interaction detection. Furthermore, an application of the concept also to other kinds of models as well as feature-oriented program slicing seems promising. In addition, we plan further experiments and improvements concerning the accuracy and efficiency of the approach.

6. REFERENCES

Comparing Program Comprehension of Physically and Virtually Separated Concerns

Janet Siegmund*  
University of Magdeburg,  
Germany

Christian Kästner  
Philipps University Marburg,  
Germany

Jörg Liebig, Sven Apel  
University of Passau,  
Germany

ABSTRACT

It is common believe that separating source code along concerns or features improves program comprehension of source code. However, empirical evidence is mostly missing. In this paper, we design a controlled experiment to evaluate that believe for feature-oriented programming based on maintenance tasks with human participants. We validate our experiment with a pilot study, which already preliminarily confirms that students use different strategies to complete maintenance tasks.

Keywords

Separation of Concerns, Program Comprehension, FeatureHouse, #ifdef

1. INTRODUCTION

Separation of concerns is an essential strategy to implement understandable and maintainable software [21]. Besides classic programming mechanisms, such as procedures and objects, many novel mechanisms for separation of concerns have been proposed in the past: components [10], aspects [14], hyper-modules [23], and so forth. Similarly, feature-oriented programming (FOP) advocates to structure software along the features it provides (i.e., user-visible characteristic of a software system) [5, 22]. That is, features are made explicit in design and code in the form of feature modules—one feature module implementing one feature.

In our field, it is common to believe that separating code along features improves program comprehension. However, program comprehension is an internal cognitive process that we cannot observe directly [15]. Thus, it is not sufficient to rely on plausibility arguments in the debate of whether some concept or mechanism improves program comprehension. Instead, we need controlled experiments to measure it [3, 8].

In this paper, we set out to evaluate whether separating features into separate feature modules improves program comprehension. In particular, we concentrate on the mechanism of FOP as implemented in the tool FeatureHouse [2]. In FOP, developers can trace each feature to one physically separated feature module. We compare the effect of physical separation on program comprehension to an implementation in which features are annotated with conditional-compilation directives such as #ifdef. We speak of virtual separation, because the #ifdef directives allow developers to trace a feature to its scattered implementation throughout the source code.

To this end, we designed a controlled experiment, in which we observe how participants comprehend source code during maintenance tasks. As material, we used two comparable software systems—one decomposed physically in terms of feature modules and one annotated with preprocessor directives. Based on experimental results, we can give recommendations on which technique of separating code along features is suitable for which task and how to improve them.

Our contributions are twofold:

- We design a reusable experiment to evaluate the impact of physical separation with FeatureHouse on program comprehension.
- We conducted a pilot study to validate the experiment and prepare a large scale run.

We plan to execute the experiment with a larger sample in the fall term. We appreciate feedback and additional research questions to evaluate. We also invite others to conduct this or similar experiments. Therefore, we provide all necessary material online at http://fosd.net/experiments.

2. PHYSICAL VS. VIRTUAL SEPARATION

To separate crosscutting concerns, several programming techniques were developed, including aspect-oriented programming [14] and FOP [22], which aim at dividing the source code into modules regarding concerns or features.

FOP as implemented by AHEAD [5] and FeatureHouse [2] separates code belonging to different features physically into separate folders, one folder per feature (and per interaction). Each folder may contain multiple packages and (partial) classes that implement the corresponding feature. To generate a product for a specific feature selection, the code from the selected features is composed, such that classes and methods that have the same name are merged by superimposition [2].

As a base line for comparison, we use virtual separation with #ifdef directives, in which features are merely mapped to code fragments with annotations in the source code. A
common mechanism is to use `#ifdef` directives in the source code to indicate which code fragments belong to which features. To generate a product for a specific feature selection, a preprocessor removes the code of all deselected features. In this approach, code belonging to a feature may be scattered over multiple classes and may be tangled with code of other features. The name *virtual* separation comes from separate tools that can create views on the source code of specific features, thus emulating modules [13]; these views are not further considered in this paper, because they deserve an evaluation of their own.

Both strategies, physical and virtual separation, allow a tracing from features to code fragments. Using physical separation, each feature can be traced to one directory, whereas, using virtual separation, we can trace a feature to multiple code locations using a global search.

To illustrate virtual and physical separation, we show example in Figure 1. Both excerpts show code from MobileMedia, a software for the manipulation of media on mobile devices [9]. On the left, we show virtual separation implemented with `#ifdef` directives; on the right, an implementation of the same code with FeatureHouse.

In prior work, we and others discussed trade-offs between physical and virtual separation [3, 12, 13, 17, 18]. Physical separation has been claimed to improve code comprehension, because, by separating features into folders, the amount of information is limited; only relevant code of a feature is present. Hence, developers might be less distracted and can focus on the code of a single feature during maintenance tasks. However, we also made the experience that (potentially due to the lack of interfaces), to understand code of a feature, base code also has to be understood. Hence, there might be important information missing, which developers have to look up in different folders. This might slow developers down compared to virtual separation, in which information of base code and feature code (but also code of other features) is present in one file. To evaluate whether physical separation of concerns indeed improves program comprehension, we designed a controlled experiment, described next.

### 3. EXPERIMENT DEFINITION

To evaluate whether physical separation of concerns à la FeatureHouse has a benefit on program comprehension, we designed a controlled experiment. To describe the settings and results, we use the guidelines provided by Jedlitschka and others [11]. To support replication, we provide all material of the experiment at the project’s website ([http://fossd.net/experiments](http://fossd.net/experiments)).

#### 3.1 Objective

With our experiment, we target the question whether participants understand physically separated source code (feature modules) different than source code that is virtually separated (preprocessor directives). To understand our research question, we need to understand how humans process information. To process information from the outside world, we use our working memory, which holds information we perceive and makes it available for further processing [4]. However, working memory capacity is limited to only few items, which are units of information, for example, digits of a telephone number or objects on a shopping list [19]. By structuring information, we can store more information. For example, we can group information of a shopping list into groceries and clothing and then memorize few items of the groceries and few items of the clothing category. In physically separated source code, the amount of information presented in one place is smaller and more clearly structured, so the working memory of participants might not be stressed too much.

However, when the present information is not enough to understand code, participants need to search relevant information. Hence, they might need more time, and during their search, they have to keep in mind where their search started. For that, they need more working memory capacity. Our first research questions are the following:

**RQ1/2:** Does physical separation of concerns improve program comprehension in terms of correctness/response time?

Additionally, we are interested in the search behavior of participants. In virtually separated code, files are larger, because they typically contain code of several features. Thus, participants might use the search function often to find information. In physically separated code, one file contains information of only one feature; hence, relevant code may be easier to find without using the search or using the search less frequently. However, the information presented in one file might not be enough to understand the code, so participants might use a global search (i.e., across modules) more often. Thus, we state a second research question:

**RQ2:** Is there a difference in the search behavior between physically and virtually separated concerns?

Furthermore, there might be a difference in the strategy participants use to find a bug. Different strategies require different amount of time and cognitive resources, so an efficient strategy can improve program comprehension. In the `ifdef` version, participants might start by using the global search function to locate code of the relevant feature, because according code is scattered across the project. In the FeatureHouse version, participants might start by opening a file in the relevant feature module, because according code is located only in that module and files are short compared to the `ifdef` version. Thus, we state a third research question:

**RQ3:** Is there a difference in the first action to find a bug?

#### 3.2 Material

As material, we use MobileMedia, which was implemented by Figueiredo and others with the help of students in Java ME with the preprocessor Antenna, which enables `ifdef` directives in Java ME code [9]. We use MobileMedia, because it was carefully developed and evaluated regarding standard coding techniques and design principles, so we can be sure to have minimized confounding effects due to badly implemented code. Furthermore, MobileMedia is often used in research to compare physically and virtually separated code (e.g., [9]). Thus, our results provide further data on the effect of physically and virtually separated code based on MobileMedia or similar systems. Of course, in future work, we need to consider additional software systems to generalize our results.

From the preprocessor version of MobileMedia, we created another version based on FeatureHouse.1 We selected FeatureHouse, because we had the opportunity to work with

1There is also an AspectJ version of MobileMedia, which uses physical separation of concerns. However, AspectJ syn-
Figure 1: Virtual and physical separation using the preprocessor Antenna (a) and FeatureHouse (b-d).

Students who are familiar with it at the same level as with preprocessors. Thus, we do not need a training session and can keep the time for the experiment as short as possible.

To ensure that both versions differ only in the underlying programming technique, two reviewers realized the refactorings. They evaluated the work of the other reviewer on few code fragments. We explicitly encourage other experimenters to evaluate the comparability of both versions and give us feedback.

An important difference between both versions is caused by the technique, such that in the FeatureHouse version, there are more folders, because for every feature or feature combination, a new folder is created, in which files are stored according to the declared packages. In the ifdef version, there are no folders for features or feature combinations, but only those folders defined by the package declarations (which are also present in the FeatureHouse version).

To evaluate our research questions, we use a between-subjects design, so we give one group of participants the ifdef version, and the other group the FeatureHouse version. This way, we can compare the performance of participants of both groups. For the first research question, we analyze response time and correctness for maintenance tasks. Response time is logged automatically, and correctness determined manually by an expert.

For the second research question (regarding the search behavior), we log how participants use the search function during solving maintenance tasks. Participants can use either a local search, that is, within a file, or a global search, that is, in all files and folders of the complete project. Both searches use strings (no pattern matching or syntactical search).

For the third research question, we log the behavior of participants, that is opening and closing files, switching between files, and using local or global search including the search term.

To control for programming experience, one of the major confounding parameters in program comprehension experiments, we apply a questionnaire to measure it [6]. Based on the value in the questionnaire, we can apply a control technique (e.g., create two groups with comparable programming experience). In addition to measuring program comprehension, the search behavior, and first action for a task, we use a questionnaire to assess the opinion of participants regarding difficulty of tasks and motivation to solve a task (both on a five-point Likert scale [16]). This way, we get more information to interpret our data.

To present source code, tasks, and the questionnaire to participants, we use the tool PROPHET [7]. It lets experimenters create tasks, specify how participants see source information to interpret our data.

3.3 Tasks

We developed five bug fixing tasks, such that we can evaluate the claimed benefit of physical separation of concerns. Hence, the class in the FeatureHouse version that contains
the bug is relatively small compared to the ifdef version. To get an impression of how short source code has to be to provide a benefit (if any), we introduced the bugs in classes of different size. All tasks were designed to have comparable difficulty, so that it does not confound the results. We encourage other researchers to evaluate the comparability of both tasks and give us feedback. Additionally, we evaluate whether comparing similar statements of different features helps to find a bug (Task 2). Furthermore, we analyze how the need to consider two classes of different features affects program comprehension (Task 5). We designed only 5 tasks to avoid a too long duration. In our experience, 2 hours is the upper limit for an experiment; after that, participants lose motivation and/or concentration, and/or become fatigued.

To present the tasks, we gave participants a bug description as a user might provide it. Additionally, we provided the feature that is present when the bug occurs, so that participants can focus on feature code. This way, we can evaluate our research question, because cohesion refers to feature code only. In Table 1, we provide an overview of all tasks. To complete a task, participants are instructed to determine the class and line number of the bug, describe why the problem occurs, and suggest a solution as verbal description. We use all information to determine whether a task was solved correctly. To measure comprehension, we analyze correctness and response time of a solution. The more correct answers and the smaller the response time of participants, the better they understood source code. Next, we describe each task in detail, show relevant code fragments with bugs highlighted, and discuss whether the FeatureHouse or ifdef version might provide benefits for comprehension.

**Task 1**

In this task, instead of setting the counter to the actual value, it is set to 0. To illustrate this bug, we show relevant source code in Fig. 2. The class that contains the bug is considerably smaller in the FeatureHouse version, such that the complete class fits on one screen. However, the original method definition in the base feature might be relevant to understand the bug. Thus, participants of the FeatureHouse group might be faster, if they do not look at the base code, or slower, if they do not look at the base code.

**Task 2**

In Task 2, a false identifier is used (SHOWPHOTO instead of PLAYVIDEO). We show an excerpt in Figure 3. Like in Task 1, the FeatureHouse version is considerably shorter. However, in the ifdef version, source code for other features (e.g., Photo) are visible, which participants might compare with feature Video and, thus, may help them to recognize that SHOWPHOTO is the wrong identifier to play a video. Another difference is the location at which the command is defined. In the FeatureHouse version, command definition and usage appears on the same screen, but not in the ifdef version. Thus, we can argue both in favor of and against a benefit for program comprehension in the FeatureHouse version.

**Task 3 and 4**

Task 3 and 4 are similar to Task 1, so we do not show source code here. In Task 3, the target is class in the FeatureHouse version is too large to fit on one screen. Thus, a possible benefit due to shorter classes might not occur here or be weaker.

**Task 5**

In Task 5, we implemented the additional feature AccessControl to observe how participants can trace source code. The feature introduces rights to manage pictures, so if users have no rights to delete a picture, they cannot delete it. As bug, we use a wrong label for deleting a picture, such that the check for according rights is never executed and a user can delete a picture without according rights (Figure 4). The definition of the correct label is in another class, so two classes have to be looked at to locate the bug. In the FeatureHouse version, the two classes are located in different feature modules, which might slow down participants.

Additionally, we designed a warming up task to let participants familiarize with the experimental setting. In this task, participants should count the occurrence of a feature (ifdef version) or how often a class is refined (FeatureHouse version). The result of this task is not analyzed.

### 3.4 Analysis Methods

To analyze the data, we use descriptive statistics (mean, standard deviation, frequencies, and boxplots) to describe response time, correctness, search behavior, and first action for a task. This way, we get an overview of how that data are
4. PILOT STUDY

To evaluate the feasibility of our design and provide some first data to evaluate our research question, we conducted a pilot study. Our participants were 8 students (graduates and undergraduates) from the University of Passau with a mean age of 23. They were enrolled in the course Contemporary Programming Paradigms, in which preprocessors and FeatureHouse were taught with comparable level of detail. Thus, participants have comparable, necessary knowledge regarding both techniques to complete the tasks. No participant was familiar with MobileMedia. All were aware that they took part in an experiment and that their performance does not affect their grade for the course. Participants volunteered to take part and did not receive compensation for their participation.

To create two comparable groups, we applied a programming-experience questionnaire a few weeks before the experiment [6]. Not all participants who completed the questionnaire showed up for the experiment. Thus, both groups differ in their programming experience. We discuss this prob-
4.1 Results

First, we evaluate program comprehension by analyzing correctness, response time, search behavior, and first action for each task to shortly address the research questions. To separate reporting data from interpreting them, we only report the data here and discuss them in Section 4.2, in which we also discuss the feasibility of our design.

4.1.1 Correctness

First, we look at correctness. In Figure 5, we give an overview of the number of correct solutions. The third and fourth task appear to be easy, because all participants found the correct solution. The first task appears to be too difficult for the FeatureHouse group, because no participant found the correct solution. The same counts for the second task for participants of the ifdef group.

4.1.2 Response Time

Second, we look at the response times. In Table 2, we show how long participants needed to solve each task and all tasks together (in minutes). For most of the tasks, the ifdef group was faster; only for the second task, the FeatureHouse group was faster. The difficulty seems to vary, because the response times differ between tasks.

4.1.3 Search Behavior

In Table 4, we show how often participants used the search feature (local, global, and combined). Participants of the ifdef group used the search considerably more often than participants of the FeatureHouse group. For the local search, participants always used it more often than the global search.

4.1.4 First Action

In Table 3, we summarize how participants started to solve a task. Participants of the ifdef group most often used a global search to find code fragments of the relevant feature, whereas participants of the FeatureHouse group most often opened a file in the relevant feature. Additionally, in tasks where a label of a button is mentioned in the bug description, some participants searched for that label. However, they did not start to search for the label in the first task where it is mentioned (Task 2), but only for the subsequent tasks. Furthermore, two participants of the FeatureHouse group started in a wrong feature (SortPhoto). We believe this is caused by the fact that also feature SortPhoto (in addition to Sorting) sounds relevant for the task.

4.1.5 Opinion of Participants

Regarding the opinion of participants, we find a tendency that the ifdef group found the tasks easier to solve, except for Task 2. For motivation, there is a tendency that participants of the ifdef group are more motivated to solve a task. This tendency might be caused by the fact that two participants of the FeatureHouse group were unhappy to be in that group (as they told us). Thus, the FeatureHouse version appears more difficult to participants and they did not like it. This can affect their performance, such that they work slower.

4.2 Interpretation

Since our sample is too small and the ifdef group is more experienced, we cannot meaningfully interpret the effect of physically and virtually separated concerns. Except for Task 2, the faster response time of the ifdef group could be caused
by the higher experience. Thus, our interpretation is only a suggestion for future experiments.

Regarding the search behavior, we found that participants of the ifdef group used the search function considerably more often than participants of the FeatureHouse group. Additionally, all participants used the local search more often than the global search. There are two interesting facets regarding the search behavior of the FeatureHouse group. First, for the second task, in which the class containing the bug consists of only few lines, participants used the global search more often. Second, for the last task, in which two classes in two different folders needed to be located to find the bug, the global search is used only half as much as the local search (similar to the search behavior for the other tasks). Thus, this tracing task seems to have comparable effort compared to the other tasks. Based on our data, we can split our third research question regarding the search behavior into three questions:

RQ3−1: Do participants of the ifdef group use more search than participants of the FeatureHouse group?

RQ3−2: Do participants of the ifdef group use more local search than participants of the FeatureHouse group?

RQ3−3: Do participants of the ifdef group use less global search than participants of the FeatureHouse group?

Regarding the first action to solve a task, participants of the ifdef group most often search for feature code with a global search, whereas participants of the FeatureHouse group opened a file in the relevant feature (or features that appear relevant). Thus, we might conclude that participants use different strategies to solve a task.

Nevertheless, we found evidence about the feasibility of our design. Participants always understood the tasks and questionnaire and knew what they had to do. Only on two occasions, participants talked to each other, but the experimenter reminded them to work for themselves. Furthermore, two participants mentioned being unhappy to be in the FeatureHouse group. Thus, when conducting the experiment, it might be useful to motivate participants of the FeatureHouse group about the benefits of FeatureHouse. However, we have to take care not to bias participants toward preferring FeatureHouse or preprocessors, because this might bias the results. Besides that, no problems occurred. Thus, the task descriptions and questionnaires seem to be clear to participants.

However, we found that two tasks (3 and 4) appear to be too easy, because all participants solved it correctly. Hence, when replicating the experiments, it might be useful to increase the difficulty of these tasks. For example, for Task 3, providing the label might have made the task too easy, because it occurs only 2 times in the complete project. For Task 4, we can provide an erroneous implementation of bubble sort, instead of a TODO in the empty method body. Furthermore, we found that one participant spent 48 minutes on Task 2. Thus, it might be useful to set a time limit for each task, for which the response times in Table 2 can be used as orientation.

5. THREATS TO VALIDITY

When designing and conducting experiments, threats to validity are unavoidable and have to be reported. In our design, several threats occur. One threat is how we obtained the modular version of MobileMedia. Basically, it was derived from the AspectJ version by refactoring. Although the refactorings and the resulting code have been reviewed carefully, it is unclear whether designing and implementing a system like MobileMedia from scratch in a feature-oriented way would have led to a different more favorable decomposition, possibly making use of more effective modularization patterns. Exploring such patterns and related anti-patterns empirically is an avenue of further work.

A second threat is caused by the sample. When comparing techniques, we have to ensure that participants have comparable familiarity with them. Otherwise, we would measure differences in familiarity, not in the comprehensibility of both techniques. To control this threat, we recruited students from a course in which FeatureHouse and preprocessors were taught. Thus, we can assume that all participants have comparable knowledge of the evaluated techniques. However, we have to be aware that recruiting students means that our results are only valid for students. If we want to draw conclusions about experts on FeatureHouse and preprocessors, we need to recruit expert programmers.

For the pilot study we conducted, our sample is too small to draw sound conclusions regarding how research questions. Thus, we used the data as evidence for the feasibility of our design rather than to evaluate our research questions. Furthermore, the FeatureHouse group is less experienced than the ifdef group and mostly unhappy to work with the FeatureHouse version. Thus, worse program comprehension of the FeatureHouse group may be caused by lower experience.

<table>
<thead>
<tr>
<th>Task</th>
<th>Group</th>
<th>Open file in base</th>
<th>Open file in relevant feature</th>
<th>Open file in wrong feature</th>
<th>Global search for relevant feature</th>
<th>Global search for label</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Ifdef</td>
<td>2</td>
<td>-</td>
<td>-</td>
<td>5</td>
<td>-</td>
</tr>
<tr>
<td></td>
<td>FH</td>
<td>1</td>
<td>1</td>
<td>-</td>
<td>1</td>
<td>-</td>
</tr>
<tr>
<td>2</td>
<td>Ifdef</td>
<td>1</td>
<td>-</td>
<td>2</td>
<td>4</td>
<td>2</td>
</tr>
<tr>
<td></td>
<td>FH</td>
<td>1</td>
<td>-</td>
<td>1</td>
<td>3</td>
<td>2</td>
</tr>
<tr>
<td>3</td>
<td>Ifdef</td>
<td>1</td>
<td>-</td>
<td>1</td>
<td>3</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>FH</td>
<td>1</td>
<td>-</td>
<td>1</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>4</td>
<td>Ifdef</td>
<td>1</td>
<td>-</td>
<td>2</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>FH</td>
<td>1</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>5</td>
<td>Ifdef</td>
<td>1</td>
<td>-</td>
<td>2</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td></td>
<td>FH</td>
<td>1</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
</tbody>
</table>

Table 3: First action participants used to solve each task.
or happiness, not the underlying technique. To avoid misinterpreting the data, we only carefully described tendencies regarding benefits and drawbacks of physically and virtually separated code.

6. CONCLUSIONS

Separation of concerns is supposed to improve program comprehension. However, there are no empirical studies that evaluate comprehensibility of physically separated code. To close this gap, we presented an experimental design to compare program comprehension of physical and virtual separation of concerns. We refactored the ifdef version of MobileMedia (virtually separated) to a FeatureHouse version (physically separated). In a pilot study with 8 students, we showed the feasibility of our design. Our next step is to replicate the experiment with a larger sample. Furthermore, we encourage other researchers to replicate our experiment. With sound empirical results, we can give recommendation how separation of concerns is suitable for which task and how separation of concerns can be improved.

7. ACKNOWLEDGMENTS

Thanks to the reviewers who helped to improve this paper. Siegmund’s work is funded by BMBF project 01M10002B, Kästner’s work partly by ERC grant #203099, and Apel’s work by the German Research Foundation (AP 206/2, AP 206/4, and LE 912/13).

8. REFERENCES

Object-Oriented Design in Feature-Oriented Programming

Sven Schuster  
TU Braunschweig  
Braunschweig, Germany  
s.schuster@tu-bs.de

Sandro Schulze  
TU Braunschweig  
Braunschweig, Germany  
sanschul@tu-braunschweig.de

ABSTRACT

Object-oriented programming is the state-of-the-art programming paradigm for developing large and complex software systems. To support the development of maintainable and evolvable code, a developer can rely on different mechanisms and concepts such as inheritance and design patterns. Recently, feature-oriented programming (FOP) gained attention, specifically for developing software product lines (SPLs). Although FOP is an own paradigm with dedicated language mechanisms, it partly relies on object-oriented programming. However, only little is known about feature-oriented design and how object-oriented design mechanisms and design principles are used within FOP. In this paper, we want to raise awareness on design patterns in FOP and stimulate discussion on related topics. To this end, we present an exemplary review of using OO design patterns in FOP and limitations thereof from our perspective. Subsequently, we formulate questions that are open and that we think are worth to discuss in the context of feature-oriented design.

Categories and Subject Descriptors

H.2.2 [Software Engineering]: Design Tools and Techniques—Object-oriented design methods; D.3.3 [Programming Languages]: Language Constructs and Features—inheritance, patterns

General Terms

Languages

Keywords

design pattern, feature-oriented programming

1. INTRODUCTION

When developing software systems, an extensible and reusable design is crucial for the durability and maintainability of the system. To achieve such a clear and maintainable structure, different mechanisms and design principles exist, depending on the used programming paradigm. For object-oriented programming (OOP), abstraction and information hiding play a pivotal role for the foundation of a clear design. On the technical side, inheritance but also interfaces are mechanisms that provide the developer with capabilities to realize different levels of abstractions. Additionally, object-oriented design patterns exist to provide general solutions for complex, recurring problems with [6].

While this is the state-of-the-art for complex, stand-alone software system, the concept of software product lines (SPL) gained momentum in recent years [4, 9]. Different approaches exist to implement software product lines, which can be divided in two categories: annotative and compositional [7]. In this paper, we focus on the emerging paradigm of feature-oriented programming (FOP), a compositional approach that extends OOP by providing reuse facilities for building product lines at large-scale. Although FOP distinguishes from OOP by specific mechanisms such as refinements for implementing software product lines, a clear and evolvable design is crucial for both approaches, FOP and OOP.

For OOP, well-established design mechanisms (inheritance, interfaces) and concepts (design patterns) exist while for FOP only little is known about design issues. However, we argue that object-oriented design mechanisms and concepts, especially design patterns, can be applied to FOP as well, because of related concepts between FOP and OOP. This, in turn, inevitably leads to several questions: Do we apply OO design patterns within FOP already (but rather implicitly than on purpose)? Is there a way to make design decisions such as usage of design patterns explicitly in FOP? Are OO design patterns applicable to FOP? What are limitations? And are there dedicated feature-oriented design patterns?

With this position paper, we want to stimulate the discussion on these (and maybe forthcoming) questions, because we believe that they are important for future work on feature-oriented design and languages. To this end, we provide a review on using OO design patterns in FOP by means of different examples. Furthermore, we point out limitations that we observed during our review.

In a broader sense, this paper also contributes to an ongoing discussion on modularity and design in FOP [8]. In this context, we stimulate discussion on the question whether dedicated feature-oriented design patterns are needed to ensure an evolvable and maintainable feature-oriented design.

Limitations: With this paper, we do not present fully-fledged and finished research results. Rather, we want to raise awareness on the role of feature-oriented design and its relation to object-oriented design (patterns). Further-
more, we focus on a specific feature-oriented approach called FeatureHouse. Finally, we rely on the exemplary design patterns presented by Gamma et al. [6], although other realizations of these patterns are possible.

2. BACKGROUND

In this section we will provide a short background on object-oriented design patterns and the paradigm of feature-oriented programming.

2.1 Object-Oriented Design Patterns

During design and implementation, it is common that certain recurring problems emerge, which have to be solved without decreasing maintainability or reusability. A design pattern is a textual description for such a common problem and its possible solution [6]. Following principles for “good” object-oriented design, patterns aim at improving the structure of a program and increasing reusability and maintainability of the source code by making it more flexible and more adaptable to changes. Examples for such design principles that are reflected by patterns are:

- favor object composition over inheritance
- program to an interface, not to an implementation
- encapsulate what varies

While different possibilities exist to realize design patterns, we here focus on the implementation and representation (using UML class diagrams) originally proposed by Gamma et al. [6]. Exemplary, we illustrate the Strategy pattern [6, p.315 ff.] by means of a class diagram in Figure 1. This pattern takes a family of algorithms and makes them interchangeable by defining an abstract strategy interface. In this pattern, the class Context holds an object of type Strategy, which provides the interface to be used. This object can be replaced with other objects of the same type, resulting in an interchangeable algorithm for the defined interface.

Figure 1: Class diagram of Strategy pattern

Design patterns are classified by their purposes into three categories of patterns: creational, structural and behavioral patterns. Creational patterns describe when and how objects are instantiated such as the Factory Method [6, p.107 ff.], which encapsulates and simplifies the creation of similar objects. The main concern of structural patterns is the composition of classes or objects, like the Facade [6, p.185 ff.], which hides the structure of a subsystem behind a new, simplified interface. Finally, behavioral patterns deal with the interaction between objects and provide dynamic behavior at runtime, like the aforementioned Strategy.

2.2 Feature-oriented Programming

Feature-Oriented Programming (FOP) is a paradigm to implement software product lines (SPL) in a compositional way [10]. Different approaches and languages exist to implement feature-oriented software product lines such as AHEAD [3], FeatureHouse [1], or FeatureC++ [2]. The core idea of FOP is to decompose a program into features. All artifacts (code and non-code) belonging to a certain feature are modularized within one cohesive unit, called feature module. A feature is an increment in functionality, visible to any stakeholder. A feature model describes commonalities and differences between the different programs of a product line and thus possible and valid combinations of features. Due to its modular fashion, FOP provides a one-to-one mapping between its implementation units (i.e., feature modules) and the features of a feature model.

In Figure 2, we show an excerpt of a stack product line implementation with FeatureHouse [1], a language-independent approach for FOP, which uses superimposition as its composition mechanism. Feature BaseStack provides the base implementation of class Stack. The two other features, Peak and Undo extend the functionality of this class. In the context of FOP, this extension or increment of functionality is called refinement. Basically, refinements offer the possibility to add or extend classes, for instance, by adding new methods or fields or changing existing ones. Methods can be composed using a specified keyword (original in FeatureHouse) to access an already existing method body. As an example, feature Undo extends method push by adding an additional statement followed by the original keyword, which invokes method push of the original class Stack. Feature Peak simply adds the method peak. To generate a program, the selected features (i.e., the corresponding source code) is composed using superimposition. For instance, if a user selects features BaseStack, Peak and Undo results into class Stack with four methods (push, pop, peak, undo) and one field.

3. COMPARING OBJECT-ORIENTED AND FEATURE-ORIENTED DESIGN

Object-oriented design mechanisms and patterns are well-understood and commonly accepted as a mean to achieve a clear and maintainable design. FOP partly relies on object-oriented concepts and mechanisms. This raises the question, how and where both approaches consolidate, especially regarding the design of the underlying programs. In this section, we present some initial thoughts on that question. In particular, we compare and contrast inheritance and refinements and discuss whether (and how) object-oriented design patterns could be applied in feature-oriented programming.
3.1 Inheritance versus Refinements

While OOP offers class inheritance as the main language mechanism to gain variability and abstraction in software design, FOP additionally offers class refinements to achieve feature modularity. In the following, we will distinguish these mechanisms.

Both, inheritance and refinements, are mechanisms to achieve code reuse and to extend classes, but beyond that, they do not have much in common. In Table 1, we provide a short distinction of both mechanisms.

<table>
<thead>
<tr>
<th>Inheritance</th>
<th>Refinements</th>
<th>Table 1: Inheritance versus Refinements</th>
</tr>
</thead>
</table>
| ... creates a new subclass to extend a class | ... extend the original class itself | class Stack {
| ... achieves variability at runtime | ... achieve variability at compile time | void push(int v) {
| ... is integrated within the language | ... are not integrated within the language | /\*...*/

We illustrate the differences between inheritance and refinement with two code examples in Figure 2 and 3, respectively. The feature-oriented implementation of Stack consists of only one class that is refined in each feature module (cf. Figure 2). Hence, for a certain variant, only one composed class exists, which contains the whole functionality of the selected features. In contrast, in our object-oriented implementation of Stack, we have to introduce a new class for every feature and every combination of features, resulting in four different classes (cf. Figure 3). In a nutshell, extending a class with inheritance always leads to a new subclass, while refinements do not have much in common.

Another difference between both mechanisms is their integration within the language and their scope. Inheritance is a language mechanism, which can be used to achieve varying behavior at runtime by creating subtypes and providing interchangeability between objects. In contrast, all variants of Stack are interchangeable, since they are subtypes of the same superclass. In contrast, refinements disappear when composing the feature modules at compile time. Hence, they allow for selecting which features and thus which refinements should be included for a certain variant before this variant is generated. Overall, inheritance and refinements can be seen as two different, orthogonal dimensions, which are rather complementing than contradicting.

3.2 Design Patterns in FOP

Since FOP and OOP share some language mechanisms, object-oriented design patterns should be applicable in feature-oriented SPLs. Furthermore, refinements should not contradict with language mechanisms used for design patterns such as inheritance or interfaces, for the previously mentioned reasons. Hence, we argue that we can use refinements to modularize design patterns in terms of features. In the following, we present examples how design patterns could be extended or modified using refinements.

In Figure 4, we show an example for creational patterns in FOP. In particular, we apply a refinement to a variant of the Factory Method (cf. Section 2.1) by providing the method createProduct(int id) with feature module Foo and using refinements to add new products. Hence, we offer the possibility of creating products of type Bar only if the feature module Bar is included. Moreover, new factory methods or whole new factories with their respective products can be introduced with new feature modules. In the same way, other creational patterns can be refined as well. For instance, the Prototype pattern [6, p. 117 ff.] can be extended using a feature module that adds new prototypes to a list of prototypes.

Structural design patterns, e.g., Facade (cf. Section 2.1), are great examples for the benefits of combining patterns of OOP with FOP. The Facade pattern hides a whole subsystem behind a simplified interface. As a result, we may use refinements to modify or extend everything within the subsystem, without interfering any other class, as long as the interface is not modified.

Behavioral design patterns such as the Strategy pattern (cf. Section 2.1, Figure 1), can be extended by new strategies via features. In Figure 5, we show the Strategy in Violet1, which is combined with the Prototype. While the abstract strategy class Graph offers the interface to grant access to the different prototypes for nodes (and edges), the concrete strategies like ClassDiagramGraph provide the corresponding prototypes. Since Violet is refactored in a very fine-grained manner, every prototype is included in its own feature module. Hence, the feature module InterfaceNode introduces the prototype for an interface node (cf. Figure 6). This leads to a one-to-one mapping of features and strategies as well as features and prototypes. We can even modularize more complex behavioral patterns using refinements. For example, in the Observer pattern [6, p. 203 ff.], the registration

---

1source code on www.fosd.de/fh
of different observers could be performed in different feature modules.

![Diagram of ClassDiagramGraph, UseCaseDiagramGraph, and StateDiagramGraph]

**Figure 5: Strategy pattern in Violet**

```java
public class ClassDiagramGraph {
    static {
        NODE_PROTOTYPES[1] = new InterfaceNode();
    }
}
```

**Figure 6: Introducing an interface node in Violet**

Since refinements are a structural mechanism, we cannot expect to change any dynamic behavior of the OO patterns. Hence, we argue, even though we are able to change the behavior of design patterns in a certain way by using refinements, we only gain advantages on a structural level.

### 3.3 Design Pattern in FOP – Use or Refuse?

Based on the review of OO design patterns and some initial insights on feature-oriented programs, we briefly address the questions that we posed at the beginning of this paper. For a more comprehensive overview, we refer to [12].

**Do we already apply OO design patterns in FOP?**

Recently, we conducted a preliminary analysis on design patterns in feature-oriented programs [12]. As a result, we detected design patterns throughout all programs, regardless whether they have been refactored or developed from scratch. Hence, we argue that design patterns are already in use with FOP. Nevertheless, a more comprehensive and quantitative analysis is necessary to make claims regarding how and which patterns are used.

**Are OO design patterns applicable in FOP?**

Based on our review and preliminary analysis of feature-oriented programs, the answer is yes. However, it is open which pattern fits very well with FOP and which do not. Furthermore, how concrete implementations look like for different feature-oriented languages has to be investigated. Another point, even discussed for OO languages, is the question whether design patterns are always beneficial or might even introduce drawbacks [5]. For instance, Smaragdakis et al. compare mixin layers, another approach for realizing compositional SPLs, with the Visitor pattern and point out certain characteristics where mixins are more advantageous than the visitor pattern [13].

**What are limitations?** From our perspective, applying behavioral patterns is limited, because these patterns focus mainly on changing behavior at runtime. Although it has been proven by Rosenmüller et al. that such patterns can be used to support dynamic binding [11], it is generally a very complex task and maybe only possible for certain languages. Furthermore, implementing design patterns with features as an additional dimension could be a complex task, especially from a programmer’s comprehension point of view.

### 4. CONCLUSION AND FUTURE WORK

Design patterns describe recurring problems (and its solution) in object-oriented design. While there is a considerable body of knowledge on design patterns in OOP, only little is known about design patterns in FOP. In this paper, we addressed this topic and reviewed exemplary (OO) design patterns from a feature-oriented point of view. We have show by example, that design patterns are applicable, but also point to possible limitations and open questions on benefits and application of patterns in FOP.

While the main contribution of this paper is to raise awareness and stimulate discussion, we determined open questions during our review of design patterns in FOP that can guide future research on this topic. In future, we want to analyze existing feature-oriented systems with respect to the occurrence of design patterns to determine whether design patterns are already used in FOP. Furthermore, the concrete realization of design patterns across different feature-oriented languages is part of our future work.

### 5. REFERENCES


[6] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. Design Patterns: Elements of Reusable Object-Oriented Software. Addison-Wesley, 1995.


Architectural Variability Management in Multi-Layer Web Applications Through Feature Models

Jose Garcia-Alonso
Centro Universitario de Merida
Universidad de Extremadura
Avda. Santa Teresa de Jornet, 38
Merida, Spain
jgaralo@unex.es

Javier Berrocal Olmeda
Escuela Politecnica
Universidad de Extremadura
Avda. de la Universidad, s/n
Caceres, Spain
jberolm@unex.es

Juan Manuel Murillo
Escuela Politecnica
Universidad de Extremadura
Avda. de la Universidad, s/n
Caceres, Spain
juanmamu@unex.es

ABSTRACT

The development of large web applications has focused on the use of increasingly complex architectures based on the layer architectural pattern and different development frameworks. Many techniques have been proposed to deal with this increasing complexity, mostly in the field of model-based development which abstracts the architects and designers from the architectural and technological complexities. However, these techniques do not take into account the great variability of these architectures, and therefore limit the architectural options available for their users. We here describe a feature model that captures the architectural and technological variability of multilayer applications. Using this feature model as the core of a model-driven development process, we are able to incorporate architectural and technological variability into the model-based development of multilayer applications. This approach keeps complexity under control whilst flexibility on choosing technologies is not penalized.

Categories and Subject Descriptors
D.2.11 [Software Engineering]: Software Architectures;
D.2.13 [Software Engineering]: Reusable Software

General Terms
Design

Keywords
Multilayer architectures, feature model, design patterns, development frameworks, model-driven development

1. INTRODUCTION

Advances in technology as well as in the use of Internet by the general population have led to ever more advanced Web applications being built. The increased complexity of these applications naturally involves greater complexity in their development. One of the principal tools that software engineers have when confronting a complex development is represented by design patterns [11] - reusable general solutions for common problems in a given context.

In the context of Web applications, one of the commonest architectural patterns is the layer pattern [5]. This pattern proposes subdividing a system into a set of layers, each of which provides services to the layer above and consumes services from the layer below. Based on this principle, complex multilayer architectures can be developed in the way that a given layer can provide services to several upper-level layers. For instance, a business logic layer can provide services to a user interface layer and a Web service layer, or a single layer can be located transversally, and thus provide services to all the other layers, an example being a log layer.

Using the layer architectural pattern, the development is divided into several layers of a lower order of complexity than that of the complete system. The development of each of these layers presents its own challenges, however, so that it is usual to go back to using other design patterns. In order to facilitate the use of those design patterns, development frameworks have become especially relevant. This is reflected in the large number of existing frameworks [25], the traffic generated by their mailing lists, and the number of job offers requesting experience in their use [23]. These frameworks "provide domain-specific concepts, which are generic units of functionality. Developers create framework-based applications by writing framework completion code, which instantiates these concepts" [3].

The use of the layer architectural pattern combined with specific development frameworks for each of the layers in the application provides greater quality in terms of reliability and maintainability [5]. But it also introduces new problems [14]. Everyone involved in building such an application must have in-depth technical knowledge in the use of the chosen frameworks. Frameworks are advanced tools whose use requires highly trained personnel. This problem can be mitigated using model-driven development techniques that abstract developers from the technical details of the technology [24].

However, current solutions are not fully optimal due to its limitations [16]. In an environment with many rapidly evolving technologies, model-driven techniques need continual updates to keep pace with technological advances. These
updates must be performed by staff with profound knowledge in three areas: the new technology to be included, the transformation language used, and the model-driven development technique or tool into which it will be incorporated. This makes it too complicated to upgrade existing model-driven techniques to cover the large number of technologies available.

To address these issues, we here present a feature model that captures the architectural and technological variability of multilayer Web applications. This model contains the commonest layers of these applications, the design patterns used in each layer, and the development frameworks that implement those patterns. Its use facilitates the architect’s work in defining the software architecture that best meets the system requirements, and in choosing the most suitable technology for the implementation.

The main advantage of this model is that it can be used as the core of a model-driven development process, as some of the currently available. This allows the developers to be largely abstracted from the technological details, an aspect that is especially important in rapidly evolving environments such as that of framework-based development, and keep complexity under control while increasing the flexibility of these processes.

The rest of the paper is organized as follows. The second section presents the background for this study. The third section describes the process used to obtain the feature model. The fourth section details this feature model and how it can be expanded to capture new or evolved technologies. The fifth section shows how the feature model can be used as the core element of a model-driven development process. The sixth section describes related work. And finally, the seventh section contains the conclusions of the study.

2. BACKGROUND

The techniques proposed by researchers in the areas of Web engineering and model-driven development have led to major advances in simplifying the work of the developers of multilayer Web applications. The use of various model-to-model transformations and code generation techniques allows these applications to be developed without the need for extensive knowledge of the technologies being used.

Nevertheless, although these techniques achieve the objective of facilitating software architect’s work, they have the drawback of limiting his or her options in designing the application’s architecture or choosing the technologies to use in the implementation. This is because these techniques impose the use of specific software architectures inherent to the models needed for the development, and the architect is not allowed to adapt them to the system’s requirements. Moreover, the architect can only choose from among a very limited set of technologies to use in the implementation. In many cases, the system requirements or external constraints to the development mean that such limitations are unacceptable.

Some of the most important of these techniques and how they limit the software architect’s options are the following:

- In [1], Acerbis et al. describe a modeling language, WebML, and a tool, WebRatio, which cover the entire process of Web application development. The work is based on the conceptual modeling of the application in two design stages: data design with which to organize the application’s information objects, and hypertext design defining the application’s interface given the preceding data design. In this way, the application is divided into two basic layers—persistence and presentation—regardless of its actual requirements. Also, the architect cannot modify this structure. Moreover, the code generated from these designs is based on ANT, XSLT, or Groovy technologies, so that the architect can not choose from the wide range of other existing technologies.

- In [17], UWE (UML-based Web Engineering), a method of model-driven Web application development based on UML, is defined. The process proposed in UWE is based on creating, in accordance with the application’s requirements, a number of models of content, navigation, processing, presentation, and adaptability. As in the previous case, these models inevitably restrict the application’s architecture. To address this problem, the authors propose using an additional technique called WebSA which we shall discuss below in the Related Work section. The code generation capabilities offered by this process use a specific framework (Spring), and the authors specifically observe that it is possible to generate code for other technologies if the corresponding transformation rules are previously available. However, although this is a valid solution with which to address the issue of ongoing technological evolution, it requires highly trained staff in both the technology to include and the techniques used to perform the transformations.

- Some studies, such as [28], increase the capacity of previous proposals in such areas as the development of RIAs (Rich Internet Applications). However, none of these improvements expands the architect’s options in designing the architecture of the application or choosing the technologies to use in development.

In some cases, the limitations of these techniques are unacceptable to some clients or projects with specific requirements of technological flexibility, such as the integration of legacy systems or in cases when certain quality factors such as high performance or stability are needed, restricting the use of such proposals. Since application architectures must conform to the corresponding requirements in the best way possible, they can not use architectures imposed by external systems. Moreover, the technologies to use in the development are often imposed by the customer’s requirements or by company policy, so that the use of a technique that imposes a very limited set of development frameworks is not really an option.

We shall here present a feature model that captures the architectonic and technological variability of multilayer applications. This feature model can be used as the core of a set of transformations that allow the architect to obtain a specific design adapted to whichever architecture and technologies are selected in accordance with the requirements of the application being developed.

3. OBTAINING THE FEATURE MODEL

The main aim pursued in the creation of this model is to incorporate architectural and technological variability into
the development of multilayer applications. Feature modeling [15] is one of the more widely accepted techniques for variability modeling [26]. In particular, our proposal uses the Cardinality Based Feature Modeling technique described in [9] since (i) it is one of the most extensively used, (ii) it is of proven utility in working with development frameworks [3], and (iii) it meets all the requirements we set out in undertaking this study. Nonetheless, it may readily be replaced by another variability modeling technique.

Since the intention was to use the feature model as the core of a model-driven development process, it had to have a well-defined structure or conform to some kind of "metamodel" so as to be treated automatically later. Such structure, however, had to be flexible enough to incorporate the large number of existing technologies. It also had to be capable of incorporating both any new technology that might arise and the evolution of existing technologies. Indeed, this was the main criterion imposed on the creation of the feature model.

To obtain a feature model that meets these requirements, we followed a bottom-up strategy. In particular, we studied a large number of development frameworks in order to extract concepts that would form the structure or "metamodel" of the feature model. For the structure to be as flexible as possible, more than 10 Java development frameworks were chosen from different developers and with different roles and goals. These frameworks were selected for being among the most commonly used within their scope [25, 23, 31]. Following is the lists of frameworks analysed: Axis, CXF, DWR, Hibernate, Ibatis, JDBC, JSF, jUnit, Log4j, PicoContainer, Spring, SpringSecurity, SpringWS and Struts.

The first architectural decision to be taken when building a multilayer application is to determine the layers of which it will be composed. Therefore, the first criterion used in analysing the development frameworks was to determine the layer or layers in which they are used.

After determining the layers that make up the application, the architect must define the design patterns to be used in implementing each of them. In particular, knowledge of which design patterns a development framework supports is of particular importance at this stage.

Finally, a framework may allow different kinds of implementation for the same design pattern. If one does not want to lose the advantages offered by the development frameworks, these different implementation techniques need to be taken into account in the feature model.

Given these considerations, we studied the frameworks listed above. The information extracted from that analysis is summarized in Table 1.

Starting from this information, we extracted the structure that the feature model would need to have. Figure 1 shows a feature model with that structure.

In general, the scope of all the frameworks studied was in a single layer. Even when the same developer supports multiple layers, this is usually implemented by being distributed among different frameworks that can be used independently. For example, the frameworks Spring, Spring Security, and SpringWS belong to the same developer, but each targets a single layer and is treated as a separate product. This, together with the fact that the layer is the main architectural pattern applied in the development of multilayer applications, the most appropriate would be for the first level of the feature model to consist of the possible layers of which an application may be composed.

Table 1: Essential framework information.

<table>
<thead>
<tr>
<th>Framework</th>
<th>Layer</th>
<th>Design Patterns</th>
<th>Implementation techniques</th>
</tr>
</thead>
<tbody>
<tr>
<td>Axis</td>
<td>Web services</td>
<td>SOAP</td>
<td>NA</td>
</tr>
<tr>
<td>CXF</td>
<td>Web services</td>
<td>REST</td>
<td>NA</td>
</tr>
<tr>
<td>DWR</td>
<td>Presentation</td>
<td>Web remoting</td>
<td>NA</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Page rearrangement</td>
<td>NA</td>
</tr>
<tr>
<td>Hibernate</td>
<td>Persistence</td>
<td>DAO</td>
<td>JPA, XML Annotations</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Ibatis</td>
<td>Persistence</td>
<td>DAO</td>
<td>NA</td>
</tr>
<tr>
<td>JDBC</td>
<td>Persistence</td>
<td>DAO</td>
<td>NA</td>
</tr>
<tr>
<td>JSF</td>
<td>Presentation</td>
<td>MVC</td>
<td>NA</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Web remoting</td>
<td>NA</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Page rearrangement</td>
<td>NA</td>
</tr>
<tr>
<td>jUnit</td>
<td>Test</td>
<td>xUnit</td>
<td>NA</td>
</tr>
<tr>
<td>Log4j</td>
<td>Log</td>
<td>Logger</td>
<td>NA</td>
</tr>
<tr>
<td>PicoContainer</td>
<td>Business logic</td>
<td>IoC</td>
<td>NA</td>
</tr>
<tr>
<td>Spring</td>
<td>Business logic</td>
<td>IoC</td>
<td>XML, Annotations</td>
</tr>
<tr>
<td>SpringSecurity</td>
<td>Security</td>
<td>Authentication</td>
<td>LDAP, OpenID, JaaS, HTTP, Authorization</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Web request, Method invocation, Access to instances</td>
</tr>
<tr>
<td>SpringWS</td>
<td>Web services</td>
<td>REST</td>
<td>NA</td>
</tr>
<tr>
<td>Struts</td>
<td>Presentation</td>
<td>MVC</td>
<td>NA</td>
</tr>
</tbody>
</table>
As mentioned above, one or more design patterns are used to simplify the implementation of each layer. Usually these design patterns are specific to each layer. Therefore, it would be appropriate for each layer present in the feature model to include the group of features representing the design patterns that can be used.

Usually development frameworks provide support for one or more design patterns. Therefore, each design pattern included in the feature model must specify the frameworks that can be used to implement it. This may result in the same framework appearing more than once in the feature model, provided that this framework supports several design patterns. This poses no problem, since the occurrence of a framework in the model implies that the framework can be used in the implementation of a particular pattern, but it does not imply that its use for a specific pattern requires the use of the same framework in all the patterns of a given layer. For example, the JSF framework supports the implementation of three design patterns – MVC, web remoting, and page rearrangement. This implies that the framework will appear thrice in the feature model. Nonetheless, the use of JSF to implement the MVC pattern in a particular application does not imply that other frameworks can not be used for the other patterns.

Finally, it is common for a framework to provide different techniques for implementing a design pattern. These techniques generally vary in the syntax used, but end up providing the same results. An example of this is dependency injection in the Spring framework. This can be done using Java annotations or using an XML configuration file. This variability aspect was also taken into account in the feature model. Where applicable, the feature model offers the techniques supported by a framework in the implementation of a given design pattern.

4. ARCHITECTURAL AND TECHNOLOGICAL FEATURE MODEL

Based on the information obtained during the framework analysis (Table 1) and the structure shown in Figure 1, we constructed a feature model that captures the variability of the frameworks considered. A fragment of that model is shown in Figure 2.

This model cannot be created solely with the information presented in the previous section, however. For the model to be representative enough for use in a development process, it must contain information not only about the frameworks but also about their interrelationships. This is because the frameworks chosen for an application development process, while independent entities, must interact and communicate with each other.

This implies that, in choosing the frameworks to be used in the development of an application, their possible incompatibilities need to be explored, and this information needs to be incorporated into the feature model. One way to include this information in the model is as constraints.

We chose OCL [21] as the language to introduce such constraints into the feature model because it has previously been used for this purpose [10], and because it is particularly well-suited to use in a model-driven process such as that which we shall be presenting in the next section.

With these constraints, we can express situations like that of when using some given framework this will prevent the use of certain other frameworks. This may be because there is no possibility of communication between them, or it may be that they are incompatible by design. For example, in the model shown in Figure 2, one can choose the Struts framework to implement the MVC pattern in the persistence layer. At that point, the JSF framework would not be a suitable choice for the implementation of any of the design patterns in the same layer. This is due to JSF being an MVC focused framework which provides only secondary support to the implementation of the other patterns. While this might not create any strict incompatibility between frameworks, it greatly complicates the communication between frameworks if JSF is not used to implement the MVC pattern. This constraint is expressed as follows in OCL (detailed information about how to express constraints in OCL can be found in [8]):

context Presentation inv:
MVC.Struts.isDefined() implies
not(WebRemoting.JSF.isDefined())
and not(PageRearrangement.JSF.isDefined())

Similarly constraints expressing the obligation to use a specific framework can be added to the feature model. Such constraints may occur when using a framework in a specific pattern that requires the use of another framework in another pattern. This may also be due to compatibility issues when a framework is compatible only with a limited set of others, or it may merely be for convenience. For example, in the model shown in Figure 2, a constraint could be included to specify that if the JSF framework is chosen for the MVC pattern then the same framework must also be chosen to implement the remaining patterns of the same layer. Constraints such as the one presented immediately below can also be included to indicate that if the SpringWS framework is chosen for implementing Web services following the REST pattern, the Spring framework must then be chosen for dependency injection. Again this is not due to any strict incompatibility between frameworks, but to the combination of these two greatly simplifying the communication between layers.

context MultiTierArchitecture inv:
WebServices.REST.Spring_WS.isDefined() implies
InversionofControl.DependencyInjection.Spring.isDefined()

Additionally, the OCL constraints could be used to include metrics about two frameworks integration level or to include qualitative aspects like the performance indicators of a framework.
With the addition of all the necessary constraints, the model incorporates all the information needed for use in a model-driven development process. In particular, one has a feature model that captures the architectural and technological variability of multilayer applications.

In view of how rapidly development frameworks evolve, one of our main concerns in constructing this model was its ease of extension to adopt new or evolved technologies. To include a new technology or a new version of an existing technology in the model, it is sufficient to include the feature for that technology in the design patterns that it supports, and to study its relationships with the other frameworks so as to add the corresponding constraints. If the new technology to be adopted requires the inclusion of a new pattern or layer, this must be added to the model before including the technology.

Additionally, the use of OCL constraints endows the model with greater flexibility. These constraints can be used not only to express the relationships between frameworks, but also to include in the model much information about the multilayer architectures. A clear example is the use of OCL constraints to specify the internal policies of a particular development company. For example, the company may have designed a standard architecture that it uses to develop most of its projects. These constraints can be used to strengthen the implementation of such an architecture, or even to reinforce the specific ways in which a framework is used to make this use uniform over all of the company’s projects.

5. A FEATURE MODEL AS THE CORE OF A MDD PROCESS

The feature model presented in the previous section is interesting in itself as a taxonomy of a set of technologies used in multilayer application development. The advantages provided by this model are really exploited, however, when it is used as the centerpiece of a model-based development process. In this section we shall discuss how the feature model, in combination with various modern model-driven techniques, can be used to guide the development process.

In this process, the software architects and designers configure the feature model in stages. "Each stage takes a feature model and yields a specialized feature model, where the set of systems described by the specialized model is a subset of the systems described by the feature model to be specialized" [9]. In this configuration process, the layers to be included in the development of the application are selected, then the design patterns used in each layer, and finally the frameworks used for the development of each pattern and the form in which they will be used.

This configuration process can be done by the architect and designers based on their expertise in the technologies involved, or it can be assisted. Such assistance is based on the use of a technique described by the present authors in [7]. This technique facilitates the choice of the patterns that best meet the non-functional requirements of the application being developed, and offers architects and designers a possible configuration of the feature model that matches the application’s requirements.

At each step in the staged configuration of the feature model, the initial design of the application is further refined in accordance with any architectural decisions taken. This refinement of the initial design is performed using model transformations. However, the rapid evolution of technologies makes it inefficient to use technology specific transformations. Instead, different model-driven techniques allow one to write transformations (e.g., high-order transformations [27] or variable transformations [29]) that can be adapted to the evolution of the technologies involved.

Performing these transformations requires additional information. Specifically, information is needed about the relationship between elements of the initial design and the selected features in the feature model. It is not always possible to obtain this information in the same way as is done to obtain information to assist in the configuration of the feature model, especially when different frameworks are used to implement patterns in the same layer.

In such cases, it is necessary to have a mechanism that allows architects and developers to specify which features should be applied to each element of the initial design of the application. This operation can be performed using the technique proposed by Arboured et al. in [4]. Their technique enables features to be related to specific elements of a model, in this case to specific elements of the initial design. Other techniques to relate feature models with other kinds of model are that represented by the Clafer language [6] and those discussed by Voelter and Visser [30]. Clafer allows metamodeling and feature modeling to be combined into a single language, and Voelter and Visser study the application of domain-specific languages in product-line engineering.

Finally, the specialization of the initial design of an application into a specific design for the chosen architecture...
and technologies is carried out, as mentioned above, using model transformations. Feature models, however, are designed for use in product lines, and are not readily applied to processes that involve model transformations specific to model-based development. In particular, the QVT language [22], the OMG standard for performing model transformations, requires the two models involved in a transformation to be based on MOF metamodels, but feature models are not based on a metamodel defined using this system.

Besides, these models would present another problem regardless of whether or not they were based on a metamodel suitable for the realization of the transformations: the feature model is not used just as it is, but different configurations of the model are also necessary. This increases the difficulty of using these models in transformations because the configurations are at a lower level in the four-layer architecture of MOF.

These problems have been resolved by Gomez and Ramos [12]. They define a metamodel for modeling CBFM feature models based on MOF. The model shown in Figure 2 can thus be converted to a model based on their metamodel. Additionally, Gomez and Ramos provide a set of QVT transformations that convert the feature model into an MOF-based metamodel. This metamodel allows configurations of the feature model to be created at the appropriate level of MOF for their use in transformation processes. A feature model based on MOF also facilitates the use of OCL constraints such as those discussed in the previous section.

The combination of the feature model presented here with the various techniques that have been mentioned in this section enables this model to be used as the core of a model-driven development process. This process simplifies the conversion of an initial design of the application (independently of which architecture and technologies are going to be used in the implementation) into a specific design for the chosen architecture and technologies. A major advantage of this process in an environment in which development frameworks evolve so rapidly is that it needs no in-depth knowledge of the technical operation of the frameworks.

6. RELATED WORK

As was noted above, there have been numerous works in the area of model-driven development which deal with the development of enterprise applications that have complex architectures, and are implemented using development frameworks. This is especially so in Web engineering, with such studies as WebML [1], UWE [17], RUX [28], etc.

These studies all provide techniques with which to richly model the functionality of Web applications, and a set of transformations that allow the user to generate the application’s code from those models. They do not, however, provide much flexibility. In most cases, applications developed with these techniques have a fixed architecture which can be neither influenced nor altered by the software architect, and which is often implicit in the models that are created. For example, most of these methods require both a persistence model and a presentation model, thus forcing the presence of these layers in any application which is developed. The case is similar in the choice of technology. The use of these methods requires the adoption of technologies for which the transformations provided in the method can generate the application’s code, thus further limiting the choices available.

Nevertheless, in this same field of Web engineering there have been proposals aimed at endowing the process of designing the application’s architecture with a certain flexibility. Melia and Gomez [18] propose an extension of the methods mentioned above which they call Web Software Architecture (WebSA). This extension adds flexibility to the previous methods by providing them with the means required to define the architecture used. Two models are added to achieve this goal: a subsystems model and a configuration model. These two models define the architectural features of the Web application that is to be developed. They are both linked to the functional models of the application (which may be constructed with any of the previous methods: OOH, WebML, etc.), and are used to generate the application with the desired architecture through a series of transformations.

A more recent study by those same authors [19] describes a similar proposal aimed at the development of RIAs. In particular, it proposes OOH4RIA as an OOH extension specifically designed for the development of RIAs. The authors describe the use of a feature model, similar to the one used in the present work, to define RIA features, and a component model to explicitly represent the architecture of the RIA. Once again, these models are used together with the functional models of OOH to generate the application’s source code.

These two works (especially the more recent) are closely related to that presented here. They pursue the same goal: to enable developers who use model-driven techniques to influence the architecture of their applications. However, the main focus of the works of Melia and Gomez is on RIA, whereas the work presented here is intended to provide support for all applications that use a multilayer architecture. Also, in contrast with the technique presented here, they do not allow developers to decide which technologies will be used for the development of the application.

In the field of framework-based software development, the studies of Antkiewicz et al. [3] and Heydarnoori et al. [13] are of particular interest. Antkiewicz et al. propose techniques that allow framework-specific designs to be modeled, and these designs then to be used to generate the source code. Heydarnoori et al. propose a technique to automatically extract framework concept-implementation templates. Unlike the work proposed here, these proposals are centred at a low level of abstraction, thus requiring their users to possess a deeper technical understanding of the frameworks they want to use. The present work raises the level of abstraction by starting from the design of the software architecture.

Finally, some works in the architecture quality field have proposed techniques to increase a system’s architectural variability. In [20], Naab and Stamm propose a technique to achieve flexibility during the architecture design stage for this flexibility to be exploitable during system evolution. This technique improves variability in the architecture, so that it can be adapted to future evolution of the system, while the technique presented here is designed to provide as much variability as possible to the architect before construction begins. In [2], Alebrahim and Heisel present a UML-based approach to modeling variability in architectures by adopting the notion of feature modeling. In their proposal, the variability is that introduced into an architecture by quality attributes, to take into account how they affect the
final architecture. In our proposal, quality attributes are used (as described in [7]) to guide the configuration process of the architectural feature model.

To the best of the authors’ knowledge, there has been no previous work on using product lines and model-driven development techniques to increase the options of software architects in the work of selecting which design patterns and development frameworks to use in the development of a multilayer application.

7. CONCLUSIONS

We have presented a feature model that captures the architectural and technological variability of multilayer applications. This model was obtained from the study of a large number of technologies. We also presented a model-driven development process which uses the feature model as its core. This model pretends to significantly increase the options available to developers when using a model-driven process to develop a multilayer application.

The following phase in this research line will be to define a method with which to incorporate into the development process the technical details of using the technologies described in the feature model. Such a model would allow one to choose the frameworks that are going to be used for a particular development. The technical details of its use, however, would have to be covered by developers who are experts in those technologies, or by transformations created by experts in those same technologies who also have advanced knowledge about transformation techniques. This situation clearly greatly complicates the incorporation of new technologies into the model-based development processes used today.

8. ACKNOWLEDGMENTS

This research was supported by the Spanish Ministry of Science and Innovation under Project TIN2011-24278, and by the Spanish Centre for Industrial Technological Development under Project GEPRODIST.

9. REFERENCES


Ensuring Well-formedness of Configured Domain Models in Model-driven Product Lines Based on Negative Variability

Thomas Buchmann  
Chair of Applied Computer Science I, University of Bayreuth  
Bayreuth, Germany  
thomas.buchmann@uni-bayreuth.de

Felix Schwägerl  
Chair of Applied Computer Science I, University of Bayreuth  
Bayreuth, Germany  
felix.schwaegerl@uni-bayreuth.de

ABSTRACT
Model-driven development is a well-known practice in modern software engineering. Many tools exist which allow developers to build software in a model-based or even model-driven way, but they do not provide dedicated support for software product line development. Only recently some approaches combined model-driven engineering and software product line engineering. In this paper we present an approach that allows for combining feature models and Ecore-based domain models and provides extensive support to keep the mapping between the involved models consistent. Our key contribution is a declarative textual language which allows to phrase domain-specific consistency constraints which are preserved during the configuration process in order to ensure context-sensitive syntactical correctness of derived domain models.

1. INTRODUCTION
Software product line engineering [10] addresses organized reuse of software artifacts. It deals with the systematic development of products belonging to a common system family. Model-driven software engineering [23] puts strong emphasis on the development of higher-level models rather than on the source code. In the past, several approaches have been taken in combining both techniques to get the best out of both worlds. Both software engineering techniques consider models as primary artifacts: Feature models [17] are used in product line engineering to capture the capabilities and the variation points of a product line, whereas UML models or domain-specific models are used in model-driven software engineering to describe the software system at a higher level of abstraction.

A key problem in software product line (SPL) development is ensuring the syntactical correctness of products. In contrast to approaches based on positive variability, which add artifacts to a common core, unused artifacts are filtered (removed) from a multi-variant set in case of negative variability. While preprocessor directives are used in source-code based approaches to build a product line based upon negative variability, additional models or metamodel extensions (e.g., profiles in case of UML) are used in model-driven approaches. It is evident that syntactical errors in derived products may result from the filtering of annotated elements although the underlying multi-variant domain model is syntactically correct. Thus, mechanisms to ensure the syntactical correctness are required in order to support automatic product derivation based on feature configurations created by the user.

2. BACKGROUND AND CONTRIBUTION
Our work is based upon the experiences we gained from the MODPL toolchain [3, 5, 6]. We realized a specific approach to ensure syntactical correctness of Fujaba\(^1\)-based UML models. A stereotype was used to annotate model elements with features. The well-formedness of products was ensured by an automatic propagation of feature annotations to dependent model elements. The toolchain was used and evaluated in a large case study described in [7, 13].

Our new toolchain FAMILE (features and mappings in lucid evolution, [8]) supports model-driven software product line engineering with a new approach to combine feature models and domain models based upon negative variability. We use a dedicated mapping model to enrich arbitrary Ecore-based multi-variant domain models with feature annotations and to finally derive products.

In this paper we present SDIRL – the structural dependency identification and repair language. SDIRL is a textual language which allows for a declarative specification of context-sensitive consistency constraints for the abstract syntax of domain meta-models. These constraints are in turn used as a basis for the automatic derivation and application of appropriate repair actions which are required to ensure the well-formedness of configured domain models. In this context, we introduce two innovative concepts – propagation strategies and surrogates – which allow the user to influence derived repair actions.

3. TOOL OVERVIEW
To put our work into context, we provide a short overview of FAMILE. Furthermore, screencasts\(^2\) demonstrating the tool and its usage and the tool’s update site\(^3\) can be found on our webpages.

3.1 Involved Models
The FAMILE toolchain comprises a meta-model and editors for feature models and configurations (see left part of Figure 1). The feature model [2] consists of a tree of features describing commonalities and differences of the software product line. Feature configurations describe the distinct characteristics of each product. In its current state, our tools support cardinality-based feature modeling.

---

\(^1\)http://www.fujaba.de  
\(^2\)http://btn1x4.inf.uni-bayreuth.de/famile/screencasts/  
\(^3\)http://btn1x4.inf.uni-bayreuth.de/famile/update/
The interconnection between the feature model and the domain model is realized by feature expressions, which are defined by our textual feature expression language (FEL). The most important language elements are covered by the screenshot in Figure 3: The boolean constants true and false return the selection states active and inactive, respectively. Features can be referenced by their name, e.g., "IEEE 802.11b", in order to make the selection state of a mapping element depend on the presence or absence of a feature in the given feature configuration. Furthermore, FEL allows for a logical combination of feature expressions, providing the keywords and, or, xor and not, as shown in the annotation VPN or SSH of the package security.
4. PRESERVING CONSISTENCY

In [3], we present consistency constraints for structural UML models as well as for behavioral models (story diagrams) built with Fujaba. Those constraints are specified formally using OCL [20]. Please notice that validation could be performed directly on the basis of those OCL constraints (by using the EMF Validation Framework, for example). However, the evaluation of OCL expressions cannot contain any side effects on the underlying model. In order to ensure the well-formedness of the configured domain model, repair actions (which do change the state of the mapping between feature model and domain model) are required and thus cannot be implemented with a validation framework. As a consequence, the OCL constraints were transformed into Java-based repair actions in a systematic but manual way in our previous approach [3, 5, 6].

However, since our new toolchain supports arbitrary Ecore-based domain models, implementing repair actions for all of them is impossible. To solve this problem, we developed the SDIRL language which is presented in this section. It allows the specification of dependency rules which impose consistency constraints on the domain meta-model using a declarative textual syntax without the need for “manual” compilation. Please note that a SDIRL document always refers to a domain meta-model. As a consequence, it can be reused for all model instances conforming to the same domain meta-model. SDIRL specifications are created using a Xtext-based text editor. For more information concerning the SDIRL editor, please refer to chapter 2 of our screencast.

4.1 Classification Of Problems

Filtering elements from a multi-variant domain model, syntactical errors that may occur can be classified into two distinct categories:

**Context-free errors** This type of error violates the hierarchical structure of the domain model. In source-code based approaches, removing a method also requires removing all of its parameters and statements, whereas in model-based approaches, the containment hierarchy defined in the meta-model must be taken into account. In EMF, a non-root object may not exist without its eContainer.

**Context-sensitive errors** This type of error violates cross-tree dependencies. Among others, context-sensitive constraints deal with relationships between declarations and applications, e.g., errors like applied occurrences of unknown types, non-existing targets of directed relationships or missing mandatory non-containment links.

In the following subsection we describe how these kinds of errors are addressed and solved with SDIRL and automatic repair actions.

4.2 Declaration Of Dependencies

While F2DMM has a built-in mechanism that allows for preserving the well-formedness of a configured domain model according to the containment hierarchy specified in the domain meta-model, context-sensitive cross-tree dependencies cannot be derived automatically from an Ecore-based meta-model. Hence, a formalism is necessary to enable domain model experts to specify those dependencies. As stated above, a validation framework is not the appropriate choice if automatic error correction should be provided.

Listing 1: OCL consistency constraint for Association ends as specified in [3]

```
1 context Association
2 inv VisibilityOfAllAssociationEnds:
3 self.featureIds()->includesAll
4 (self.memberEnd.featureIds() asSet())
5 asSet()
```

In our previous approach [3, 5], we manually implemented repair actions as so called feature propagation rules. The application of those rules leads to a configured domain model which satisfies all well-formedness constraints. Listing 1 shows the OCL constraint which was used for associations and their corresponding ends [3]. A set of feature annotations may be assigned to an element of the domain model with the help of an UML stereotype. Set members are implicitly connected via conjunction. The constraint implies that the set of feature annotations assigned to member ends of an association must be a subset of the feature annotations of the association itself. In our previous approach, we manually implemented
methods that propagated feature annotations to required model elements in order that the constraint always holds for configured domain models.

Within the F2DMM tool, SDIRL provides a declarative language that allows for a concise specification of cross-tree dependencies in Ecore-based domain models. No “manual” implementation effort for derived repair actions is required. Dependency rules phrased with SDIRL are evaluated during the mapping process. In case of conflicts, automatic repair actions are derived and applied to ensure the well-formedness of the configured domain model (see next subsections). As an example we use UML class diagrams as domain model in this section, because we assume that the reader is familiar with them.

Listing 2: SDIRL rule for Association ends

```
1  dependency AssociationMemberEnd {
2      element assoc : uml.Association
3      requires target : uml.Property = {
4          assoc.memberEnd
5      }
6  }
```

Listing 2 shows a SDIRL rule for UML associations. A dependency rule between an association and its member ends is defined. The keyword element introduces a variable for domain model elements which depend on other domain model elements via a non-containment reference that is specified using an OCL expression in the requires statement. After evaluating the expression, the result is bound to the variable defined at the beginning of the requires statement.

Once a mapping model is opened in the F2DMM editor, each SDIRL dependency rule is evaluated for each mapping with compatible type. A dependency is recorded between the context model which contains a parameter also contains its required type. The when constraint ensures that the rule is not applied to return parameters, because they may also have the type void, represented by an unset type reference.

### 4.3 Surrogates

Filtering elements might result in unintended information loss. Let’s take generalizations in a UML model as a prominent example. Within the UML specification [19], a generalization is defined as a directed relationship. It is associated to the specific classifier via composition while the general classifier is referenced with the help of directed non-containment reference (see Figure 5). As a consequence, filtering a superclass will violate a well-formedness constraint, since the result would be a dangling edge. Thus, the generalization was filtered as well in our previous approach. But this behavior might be too strict in some cases. Given the fact that the filtered class was part of an inheritance hierarchy, the user might want to replace the filtered referenced class with its closest non-filtered superclass, for example.

To address this challenge, we introduced the concept of surrogates in SDIRL (see Listing 4). Using this mechanism, filtered targets of non-containment references, mapped by the element bound to the requires variable, may be replaced with objects of the same target type. A dependency declaration can include an arbitrary number of surrogate statements, where OCL expressions that must conform to the type of the requires variable can be phrased. The expression can refer to the objects bound to the element and requires variables again. Objects that result from evaluating this expression are recorded as so called surrogate candidates. In case of our generalization example, these are all superclasses of the required class (returned after evaluating cls.allParents()).

Listing 3: SDIRL rule for applied occurrences of filtered types in method parameters

```
1  dependency ParameterType {
2      element param : uml.Parameter
3      requires type : uml.Type = {
4          param.type
5      }
6      when {
7          param.direction <> uml::ParameterDirectionKind::return
8      }
9  }
```

SDIRL allows to restrict the establishment of a dependency relationship with an optional when constraint. After binding objects to the element and requires variables, the boolean when clause is evaluated. This OCL expression can refer to the values bound to both variables. In case the condition does not hold, the dependency between the current pair of objects is discarded.

Listing 3 considers method parameters as an example for applied occurrences of types, which may be filtered during the product derivation step. In the UML specification, a parameter is a subclass of TypedElement. A TypedElement is connected to a corresponding Type via a directed non-containment reference (see Figure 5). The SDIRL rule shown in Listing 3 defines a dependency between a parameter and its type. As a consequence, each configured domain model which contains a parameter also contains its required type. The when constraint ensures that the rule is not applied to return parameters, because they may also have the type void, represented by an unset type reference.
4.4 Repair Actions

In the following, we will describe how SDIRL rules are evaluated in order to ensure well-formedness during the product derivation process. Figure 6 depicts four different steps involved in that process. As a simple example, the figure shows an UML containment tree which resembles the mapping in Figure 3: a package contains three different classes. One class contains a property, the other one contains an operation. Furthermore, generalizations between the classes form cross-tree dependencies.

In a first step, dependencies for each mapping model element are calculated based on (a) the containment tree and (b) SDIRL rules for cross-tree dependencies. In the example, containment dependencies are established between each class and the parent package as well as between each property or operation and its containing class. Additionally, two generalizations, represented by arrows, are contained by and thus require their respective class. Additionally, two generalizations, represented by arrows, are contained by and thus require their respective class. The first step also comprises the pre-calculation of surrogate candidates. The target of the upper generalization can replace the target of the lower.

In step two, feature expressions are evaluated with respect to the current feature configuration. The resulting selection states are assigned to mappings (see Figure 4). During the third step, dependency conflicts are detected and resolved. A dependency conflict occurs whenever a pre-calculated dependency contradicts the selection states of the involved mappings. This applies if the following conditions are true:

- Mapping A requires mapping B due to pre-calculated SDIRL or containment dependencies (A \rightarrow B in our notation\(^4\)).
- The selection state of mapping A is active or enforced.
- The selection state of mapping B is inactive or suppressed.

It is obvious that a domain model element included in a product may not require another element which is excluded by configuration. In case of a containment hierarchy, the contained object requires its respective container, e.g., a class cannot exist without its containing package. To resolve such conflicts, the selection states of the respective mapping elements is changed. For that purpose, the user can choose between two different propagation strategies (see Figure 7). Based upon the chosen strategy, selection states are propagated either from or to required elements in case a conflict is detected.

**Forward propagation** The conflict is resolved by artificially excluding the context mapping A (selection state suppressed). Consequently, the selection state of mapping B is propagated in the direction of the dependency arrow (forward direction).

**Reverse propagation** The selection state of mapping B is artificially made positive (enforced). A's selection state is propagated against the direction of the dependency arrow (reverse direction).

Similar strategies are applied to enforce or suppress elements in selection state pending (see Figure 8). This reduces the need of redundant annotations for a hierarchy of mappings gradually depending on each other. For instance, annotating a UML package propagates the resulting selection state to each of the package's contents. While forward propagation can only result in suppressed mappings and reverse propagation in enforced elements for conflict resolution, propagating the state for mappings without annotated feature

---

\(^4\)The inclusion of \(B\) is a necessary condition for the inclusion of \(A\).
expression can result in both artificial selection states for each strategy.

In Figure 6, forward propagation is applied in order to resolve dependency conflicts. For instance, the middle class is decorated with an inactive overlay. Its contained operation has a positive annotation, resulting in a dependency conflict: The active operation requires its containing inactive class. After applying forward propagation, the contained operation has the selection state suppressed. Contrastingly, reverse propagation would have enforced the selection state of the containing class.

The upper generalization is also assigned the selection state suppressed because of a containment relationship (a generalization is always contained by the specific classifier, see Figure 5). Due to the pre-calculated dependency which resulted from an application of SDIRL rule 4, the generalization between the most specific class and the class in the middle is also decorated with the selection state suppressed. This ensures a syntactically correct mapping model after having completed step three: No conflicts occur either on containment or SDIRL-defined dependencies. Repair actions are demonstrated in chapter 5 of our screencast.

4.5 Well-Formedness Of Products

During product derivation (step four), all suppressed and inactive elements are filtered from the multi-variant domain model (see chapter 8 of our screencast). Elements with selection state incomplete are only filtered if the user has chosen the corresponding option. Additionally, surrogate candidates pre-calculated in the first step may replace applied occurrences of filtered elements as described in subsection 4.3. The target of the generalization originating from the most specific class is surrogated – the third indirect selection state – by the super class of the inactive class in the middle. Please note that repair actions are not persisted within the mapping model, instead they are performed at runtime before a specific feature configuration is applied to derive a product.

Our tool supports three different methods for choosing one of the surrogate candidates in question: In a fully automatic mode, the first matching surrogate candidate is selected whereas in an interactive mode the user can select among the set of all candidates. Furthermore the user can choose not to use surrogates at all. In case all potential surrogate candidates are excluded, or candidate selection is skipped by the user, the surrogated element will be excluded from the configured domain model.

In Figure 3, the mapping of a part of the example UML model has already been presented from the editing perspective. The application of propagation strategies has been described above. The left part of Figure 9 shows the relevant part of the domain model in concrete UML class diagram syntax. For a better understanding, the classes have been colored afterwards according to their mappings’ selection states. At first glance, the mapping is not consistent: The class IEEE802_11aConnector requires its superclass AbstractIEEE802Connector, which is excluded due to a negative feature expression. The SDIRL declaration GeneralizationTarget, however, can make sure that in case of an exclusion of AbstractIEEE802Connector, the base class AbstractWifiConnector becomes the new generalization target. The result is depicted in the right part of Figure 9: The most concrete class directly inherits from the most abstract class, omitting the middle inheritance layer.

5. RELATED WORK

In this section we focus on mapping approaches only. There are several other model-driven product line approaches [14, 25], but they are based on an extension of UML to express variability directly in the domain model rather than on mapping elements of a distinct feature model to a domain model. Due to space restrictions, the approaches mentioned above are omitted in the discussion below. Furthermore, we will focus our comparison on the ability of automatically detecting and correcting errors that contradict well-formedness constraints of configured domain models.

The tool fmp2rsm³ combines FeaturePlugin [1] with IBM’s Rational Software Modeler, a UML-based modeling tool. The connection of features and domain model elements is realized by embedding the mapping information in the domain model using stereotypes (each feature is represented by its own stereotype). The visibility of domain model elements in corresponding configurations is determined by so called presence conditions [11]. Several constraints are given to detect erroneous configurations. The authors use explicit and implicit presence conditions to preserve the well-formedness of the configured domain model. They specified those conditions for UML class diagrams and activity diagrams. Our approach provides a more general solution as SDIRL allows for a declarative specification of dependency constraints for arbitrary Ecore-based domain models. Furthermore, the user can toggle the propagation strategy, while fmp2rsm seems to be restricted to what’s similar to our forward propagation strategy.

FeatureMapper [16] is a tool that allows for the mapping of features to Ecore-based domain models. Like our tool, it follows a very general approach permitting arbitrary EMF models as do-

³http://gsd.uwaterloo.ca/fmp2rsm
main models. FeatureMapper provides basic capabilities of checking well-formedness constraints of the Ecore meta-model, but it does not provide automatic repair actions. In [15], the author lists several consistency constraints which have to be met to ensure the well-formedness of the overall product line. In the discussion section he states that well-formedness of configured domain models can be ensured in case that well-formedness rules for the target language (e.g., UML) exist. In this paper we presented an approach that allows to specify those rules easily for arbitrary domain models and to automatically derive repair actions which are applied during the product derivation phase.

VML* [26] is a family of languages for variability management in software product lines. It addresses the ability to explicitly express the relationship between feature models and other artifacts of the product line. It can handle any domain model as long as a corresponding VML language exists for it. VML* supports both positive and negative variability as well as any combination thereof, since every action is a small transformation on the core model. As a consequence, the order in which model transformations are executed during product derivation becomes important. While our approach provides automatic repair actions using different propagation strategies, in VML* the SPL developer has to deal with this problem without any further support.

MATA [24] is another language which also allows to develop model-driven product lines based on UML. It is based on positive variability, which means that around a common core specified in UML, variant models described in the MATA language are composed to a product specific UML model. Graph transformations based on AGG [22] are used to compose the common core with the single MATA specifications. However, during the product derivation process, the order in which the single model transformations are carried out is crucial. Executing the transformations in a wrong order may result in syntactical errors in the configured UML model. Specifying the correct order is the modeler’s task and in contrast to our approach no tool support is provided in terms of automatic repair actions.

CIDÉ [18] is a tool for source-code based approaches. It provides a product specific view on the source code, where all source code fragments which are not part of the chosen configuration are omitted in the source code editor. The approach is similar to #ifdef-preprocessors. The difference is that it abstracts from plain text files and works on the abstract syntax tree of the target language instead. Using this mechanism, two rules are used to ensure context-free syntactical correctness in a language independent way: only elements which are declared optional may be filtered, and conforming to a subtree rule, all child nodes must be removed in case the corresponding parent is filtered. The authors of [18] provide a sample implementation for Java. Support for additional programming languages can be added to CIDÉ by specifying a so called FeatureBNF grammar. Our approach does not only provide the correction of context-free syntactical errors (which can be derived via the containment hierarchy of the domain meta-model), but also the correction of context-sensitive errors (based upon a SDIRL document for a domain-specific meta-model).

In our previous work [6, 3, 5] we used the UML profile mechanism to annotate domain model elements. As a consequence, the mapping information was persisted within the domain model. Enforced and automatic feature propagation in a bottom-up way was employed by manually implemented repair actions. In our current approach, SDIRL specifications are interpreted by F2DMM to derive repair actions automatically. In addition, the modeler can now choose between two propagation strategies. Furthermore, surrogates can be used within repair actions either in an automatic or interactive mode. Previously, annotations were propagated to depending elements to ensure the context-sensitive syntactical correctness of the configured domain model. In our current approach, selection states rather than feature annotations are propagated.

6. CONCLUSION

In this paper, we presented an approach to ensure the well-formedness of configured domain models for model-driven product lines based on negative variability. F2DMM is the evolution of our previous work and supports the mapping of features to arbitrary Ecore based domain models. F2DMM does not depend on the concrete syntax of a DSL and represents the mapping model as a tree which reflects the structure of the domain model. Domain-specific consistency constraints can be phrased in our textual SDIRL language. These constraints are evaluated in order to detect dependency conflicts, which are resolved by the propagation of selection states, letting the user choose one out of two available strategies. When deriving a product, the concept of surrogates helps the user minimize the information loss eventually produced by propagation.

6.1 Lessons Learned

Our version of the HAS example constitutes an artificial case study designed to demonstrate the core concepts realized by F2DMM, such as surrogates or propagation strategies. Furthermore, we learned our first lessons from porting the MOD2-SCM case study [7, 13] to our new toolchain. To this end, a SDIRL document has been defined for UML class diagrams (7 dependency rules, 2 surrogate rules) and state charts (2 dependency rules). Specifying these rules was an easy task as we only had to adopt the OCL statements identified in our previous approach [3, 5]. Originally, those were obtained from interpreting the UML specification [19].

In terms of scalability, SDIRL has the advantage that it does not need to scale with model size, but with meta-model size, as dependency rules are reusable for different instances of the same metamodel. This is why we plan to extend our existing SDIRL specification for other UML diagrams like package diagrams, use case diagrams and activity diagrams. The current version of the SDIRL document is hand-written and contains dependency specifications for the diagrams currently supported by our UML-based modeling environment Valkyrie [4]. We will check if the missing dependency constraints can be automatically derived from the XMI sources provided by the OMG.

6.2 Future Work

At the moment, the ported case study comprises structural modeling with class diagrams only. In a next step, we plan to port the behavioral models of MOD2-SCM which are specified using Fujaba’s story diagrams. The story diagrams will be replaced by ModGraph [9] specifications. ModGraph is an Ecore-based language which supports graph transformations for behavioral modeling for EMF. To provide a better usability, we plan to work on a better integration of F2DMM into the concrete syntax of Ecore-based model editors. Furthermore we are looking for new case studies to further evaluate our approach.

Acknowledgements

The authors want to thank Lutz Lukas for implementing the feature modeling tool during a master project. Furthermore, we thank Bernhard Westfechtel for the valuable inputs on the draft of this paper.

4http://www.omg.org/spec/UML/20080501/Superstructure.xmi
7. REFERENCES


Supporting Multiple Feature Binding Strategies in NX

Stefan Sobernig  Gustaf Neumann  Stephan Adelsberger
Institute for Information Systems and New Media
WU Vienna
Austria
{firstname}.{lastname}@wu.ac.at

ABSTRACT
Feature-oriented programming (FOP) toolkits restrict implementers of software product lines to certain implementation choices. One is left with the choices between, for example, class-level or object-level extensions and between static or dynamic feature bindings. These choices are typically made at an early development stage causing an unwanted lock-in. We present a feature-oriented development framework based on dynamic, object-oriented constructs for deferring such design decisions by piggybacking on first-class language entities (metaclasses, mixins). A framework prototype is available for the scripting language NX. NX provides the required object-oriented language infrastructure: a reflective language model, metaclasses, multiple class-based inheritance, decorator mixins, and open entity declarations. We exemplify the approach based on a Graph Product Line.

Categories and Subject Descriptors
D.3.2 [Programming Languages]: Language Classifications—Object-oriented languages, Scripting languages; D.3.3 [Programming Languages]: Language Constructs and Features

General Terms
Design, Languages

Keywords
Dynamic software product lines, feature-oriented programming, dynamic feature binding, static feature binding

1. INTRODUCTION
A software product line (SPL) provides a common code base for a family of related software products and a product line model (e.g., a feature model) specifying the set of valid products which can be built from the product line. An important approach to constructing software product lines in an object-oriented (OO) programming environment are collaboration-based designs [23].

In a collaboration [1], objects and classes interact by exchanging messages to realize an integrated piece of functionality. The base product is a collaboration implementing a domain model using a mix of OO composition strategies (e.g., a structure of associated objects and classes). The implementation of a feature is a code unit that is designed to extend the collaborations of the base product. The target products (the instances of the SPL) are built from a set of software assets comprising the base product and feature implementations selected by a valid configuration of the product line model. Therefore, the composed products embody the collaboration-based designs of the base product and of the feature modules. This step of building an SPL instance based on feature composition is supported by object-level composition techniques as well as by dedicated approaches for feature-oriented programming (FOP; [23]).

In order to implement a software product line, various design decisions must be made on the base product and on the individual feature implementations. Important decisions are: How is the set of assets organized into separate code units to be combined into a final product? How can feature cohesion [9] be achieved for these code units? How should object collaborations and their feature-specific refinements be expressed and made explicit? Which OO extension mechanisms could/should be used for feature composition? Is first-class code representation (data structures, objects, classes) of the assets required? Should the target product be a code unit (such as a class) that should be further reusable and refinable? What is the desired feature binding time? In other words, in which program phase are the feature implementations included into the product: at design time, at compile time, at start-up time, or at runtime? What is the desired/required product granularity: Is the product to be represented as a collaboration of objects or of classes?

By adopting certain object-compositional techniques or a particular FOP toolkit, many of these decisions must be made comparatively early in an SPL development cycle. For example, the chosen FOP approach (e.g., rbFeatures [6]) and the underlying OO language (Ruby) determine certain decisions due to the programming language’s OO model. For Ruby, this model is class-centric with single, class-based inheritance and a form of mixin composition based on dynamic superclass injection. The FOP approach might also provide mandatory abstractions for features (e.g., feature classes) and products (e.g., product classes). Collaborations of classes and objects could be expressed in a language-supported manner (e.g., by declaring a namespace per collaboration). As for the feature composition, the toolkit...
might adopt a static, class-level approach (e.g., by generating a composed source representation of a class structure at the SPL build time).

These decisions appear to be prematurely taken as they come bundled with the chosen FOP framework and the underlying programming language. After having implemented the product line to a large extent, revisiting any of those decisions at some later time (e.g., due to changed requirements) might even require a complete re-implementation in a more fitting FOP environment.

In this paper, we present an FOP framework based on dynamic OO constructs that allows for deferring the design decisions such as the feature binding time and the product granularity. To demonstrate the feasibility of the approach, we implemented the framework prototypically in the dynamic, object-oriented scripting language NX [11]. This prototype showcases the required OO infrastructure for dynamic software evolution [17], comprising a reflective language model, metaclasses, multiple class-based inheritance, and composable inheritance hierarchies [15]. With this, our approach provides means to defer decisions about ...

- the representation of feature modules,
- the feature binding time (static, dynamic), and
- the composition granularity (class, object).

The remainder of this paper is structured as follows: In Section 2, we elaborate on our motivation to support variable SPL implementation decisions and we identify a set of challenges. In Section 3, we focus on the necessary language support to address these challenges, and introduce a lightweight realization based on the scripting language NX. Then, we briefly compare our approach with related work on SPL implementation techniques (Section 4). We close with a summary and an outlook (Section 5).

2. VARIABILITY IN FEATURE COMPOSITION DECISIONS

In the following, we will use the common example of a Graph Product Line (GPL) to illustrate our approach. This product line example has been used in closely related work on FOP [20, 16] and will, therefore, facilitate comparing the approaches. As a product line model, the GPL is shown as a feature diagram in Fig. 1. The GPL is modeled as a family of products which implement different types of graphs (colored, weighted, directed, undirected, edge-labeled, etc.), different representation strategies (e.g., adjacency or incidence lists), and support algorithms (e.g., for graph traversals). For this paper, we only look at selected features. The feature colored adds coloring support to graph edges. The second feature, weighted, adds labeling support to graph edges to store and to attach weightings to edges. As shown in the feature diagram in Fig. 1, these two exemplary features are both optional (depicted by the empty dot markers) and simultaneous. That is, this product line model allows for four valid products: plain graphs, colored graphs, weighted graphs, and, both, colored and weighted graphs.

The design of the exemplary GPL is collaboration-based and layered (see Fig. 1, on the left). The graph base product is implemented by a collaboration of three entities: Graph, Edge, and Node. The two feature implementations of colored and weighted refine these base entities (i.e., Edge) in an incremental manner (e.g., by adding to the printing facility of the edges). For such designs, numerous feature implementation techniques have been proposed, for example: mixin layers [23], delegation layers [16], virtual classes [2], and decorator layers [19].

2.1 Feature Binding and Composition Levels

Including a feature into a program is referred to as binding a feature. Feature binding can occur at several binding times, with each programming language and runtime environment providing a characteristic set of binding times (pre-processing time, compile time, load time, program execution time), and in certain binding modes (i.e., fixed, changeable, or dynamic; [3]). Static feature binding occurs at an early binding time and represents an irrevocable inclusion of a feature into a program. Forms of dynamic feature binding allow for deferring feature inclusion to later binding times (e.g., during program execution) and for revoking the inclusion decision during the lifetime of a program.

Composing feature implementations can be performed at different levels of abstraction: the object, the class, the method, the sub-method, or the statement levels. For the scope of this paper, we limit ourselves to objects and to classes (see Fig. 2), the primary abstraction levels in object-oriented, collaboration-based designs [23]. At the class level, the derived product is represented by a single composed class or a composed, collaborative class structure to be instantiated. At the object level, the product is embodied by a single composed object or a composed object collaboration.

![Figure 1: A Collaboration-based Design (left) and a Feature Diagram of the GPL (right)](image)

![Figure 2: Variable feature composition](image)

Along these two dimensions, FOP approaches [20, 16, 23] fall into four categories (Fig. 2). For example, the code generation approach of Rosenmüller et al. [20] covers static class-level (FeatureC++ compound classes) and dynamic object-level feature compositions (FeatureC++ decorator layers). By offering multiple feature binding strategies, an FOP environment realizes composition variability:

- **Changeable feature binding times** This allows the product line implementer to derive products from one code base which can benefit either from static feature binding (e.g., allowing for code-level optimization, avoiding binding overhead in execution times, minimizing the memory footprint, removing otherwise dead code) or from dynamic feature binding (e.g., product reconfiguration during runtime,
lazy acquisition of platform-specific product refinements).

Changeable composition levels. Closely related, one might learn that certain feature implementations should only be applied to selected product instantiations (depending on runtime conditions). Class-level implementations of product line assets represent a family of product instantiations. To obtain a handle on one of these instantiations (i.e., an instance), idioms such as singleton classes to represent product instances at the class level must be devised (see, e.g., [2]). Alternatively, an FOP framework might support a transition to an object-level composition.

Mixed binding/composition strategies. Finally, consider the requirement to mix different binding strategies and composition levels. First, a product is composed statically at the class level (e.g., through source code generation), resulting in a class collaboration composed from a base product and certain features. To enact the collaboration, instances of the collaborating classes must be created. This instance structure could be further extended dynamically and at the object (instance) level by another feature during runtime, independently from further instantiations of the product’s class collaboration. This requires the flexibility to change the composition strategy at arbitrary times (and not only at SPL build time [19]).

2.2 Challenges

A survey of related work [19, 7, 1, 16, 23, 24] reveals important challenges of providing composition variability:

Single code base. At the times of designing and of implementing the product line assets (e.g., the core and the feature modules), adopting several level/binding combinations should not require the redundant and diverging implementation of features [19]. Code specific to a given feature, bindable both statically and dynamically (e.g., weighted in the GPL), should not be kept in two different implementation variants. Also, the feature implementations should not contain binding-specific boilerplate code (e.g., wrapper code for feature-module loading at runtime). If neglected, there is the risk of introducing code clones [22].

Avoiding decomposition mismatches. In an object-based decomposition of a layered, collaboration-based design, the collaborations (Graph, colored, and weighted in Fig. 1) and the collaboration parts (Graph.Edge, weighted. Edge) are represented by distinct objects. For the client of a collaboration-based product instantiation (a weighted graph object), the parts of a complex collaboration form single conceptual entities (e.g., the composed, most refined edge kind). This decomposition mismatch can entail a self-problem during method combination and method forwarding [8, 16].

Composition locality. Critical operations (e.g., message sends) on and within a composed collaboration (a weighted graph) should be local to the composed collaboration. The composed collaboration so sets the context for, e.g., constructor calls [24, 16].

Symmetry: Binding and unbinding. For feature bindings to be fully dynamic, the binding operation must be reversible [7]. This is also necessary to form valid products during runtime when facing mutually exclusive features.

Product-bounded quantification. For binding feature implementations, quantification [5] refers to evaluating selection predicates over a program structure (an AST, an interpreter state) to match code units (objects and methods in the base program) for performing transformation, weaving, and intercepting operations on them. Support for dynamic feature binding allows one to create multiple products (and product instantiations) side-by-side [19]; for example, multiple graph products each with a different feature configuration. This requires the client code to manage multiple feature compositions. Reconfiguring selected products (e.g., unbinding the colored feature) must preserve the feature composition of the remaining products through tailorable quantification statements.

Host language integration. The product code resulting from a feature composition step should be usable directly from native applications written in the host language. As an example, consider the plain C++ support as discussed in [19]. Any unwanted interactions between the FOP infrastructure (e.g., collaborations) and the host language features used to implement them (e.g., classes and class inheritance systems, the type system) must be controlled [23]. For example, if the product derived was represented by a collaboration structure implemented by a set of (nested) classes, these classes would have to remain refi nable by means of native subclassing (the inheritance hierarchy) without breaking the collaboration semantics (the extension hierarchy).

3. LANGUAGE SUPPORT FOR VARIABLE FEATURE COMPOSITION

In this section, we present an approach to supporting both static and dynamic, as well as class- and object-level feature composition. The approach adopts established high-level, object-oriented abstractions for object composition (i.e., decorator mixins, class-based multiple inheritance), object/class aggregation, and metaclasses. While applicable to several language environments providing these constructs, we showcase the approach for the dynamic, object-oriented scripting language NX [11] because it provides built-in support for all of these composition operations. Therefore, NX is a convenient test bed for an implementation study.

3.1 The Scripting Language NX

NX is a highly flexible, Tcl-based, object-oriented scripting language. NX is a descendant of XOTcl, a language designed to provide language support for design patterns [13]. The object system of NX is rooted by a single class: nx::Object. All objects are instances of this class. In NX, classes are a special kind of object providing methods to their instances and managing their life-cycles. These class objects (simply classes, hereafter) are instances of the metaclass nx::Class. NX supports object-specific behavior: Objects can carry behavior distinct from the behavior specified by their class. This behavior can be defined in object-specific methods and by decorator mixins (see per-object mixins in [12]). The object system is highly flexible, the relations between objects and classes and among classes can be changed at arbitrary times. NX supports dynamic software evolution [17] by supporting dynamic state and behavior changes at runtime, as well as dynamic changes to program structure and to program composition [10]. In the remainder, we concentrate on the language features of NX relevant for supporting feature-oriented programming. Throughout the section, we refer to the collaboration-based design of a Graph Product Line (GPL, see Fig. 1).

Creation of Objects and Classes in NX. Fig. 3a shows the base classes nx::Object and nx::Class with a subset
Collaboration concept as an NX metaclass (lines 1–2). A metaclass is a specialized \texttt{nx::Class}. The metaclass will get more behavior later. Then, the NX class \texttt{Graph} is defined as an instance of the \texttt{Collaboration} metaclass (lines 4–20). This collaboration contains the child classes \texttt{Graph::Edge} and \texttt{Graph::Node}. Note that these class names are prefixed by the name of the actual collaboration \texttt{Graph}. The collaboration class \texttt{Graph} can be used like any ordinary NX class: It can own properties and methods (see lines 5–10 in Fig. 4b), it can be instantiated (see line 22) and subclassed.

In the UML, the collaboration class \texttt{Graph} is represented using a UML class stereotyped «\texttt{Collaboration}» and an attached, equally named UML package (see Fig. 4a). The containment relation between the collaboration class (\texttt{Graph}), the package (\texttt{Graph}), and the nested classes (e.g., \texttt{Graph::Edge}) is modeled using the cross-hair notation $\odot$.

### 3.2 Variable Feature Composition in NX

Below, we define the code assets of a SPL to be used as the single source for static and dynamic feature binding. The same assets are also used to compose products at the class level and at the object level. Furthermore, we show how to combine these feature composition techniques. Moreover, the implementation techniques honor the previously identified requirements (see Section 2.2). We outline two techniques for dynamic feature binding at the object level and at the class level, respectively. Then, we elaborate on turning dynamically composed product representations into their source representations to be used for static feature binding. Our approach differs from prior approaches in two respects: First, in a dynamic scripting environment as NX, dynamic feature binding is the native mode. Second, we support all four combinations of composition levels (object, class) and binding modes (static, dynamic) while existing approaches are mostly limited to two: static class-level and dynamic object-level bindings [19].

#### 3.2.1 Common Assets

In a first step, we create the code assets of the Graph Product Line (GPL) as aggregated objects. The assets consist of the collaboration implementing a basic graph and the feature modules (\texttt{weighted} and \texttt{colored}; see Fig. 5). This allows us to address and to handle the assets as objects in our minimal FOP framework. As objects, the product line assets can be easily introspected and modified using...
standard programming idioms. The collaboration classes (Graph in Fig. 5) are both class objects and namespaces. As namespaces, they add namespace qualifiers (Graph::*) to disambiguate the objects representing collaboration parts (e.g., Graph::Edge vs. weighted::Edge). As objects, they provide a collaboration interface to client objects. Most importantly, the collaboration interfaces expose factory methods (new edge(), new node()) to instantiate refined, collaboration-specific variants of the contained objects. The generated factory method supports composition locality for clients (see Section 2.2).

Likewise, we define feature module classes as specialized collaborations (see line 1 in the listing above). In Fig. 5, the corresponding UML classes are tagged as «FeatureModule». In contrast to collaboration classes, feature modules are not meant to be instantiated directly. Feature modules represent intermediate and abstract collaborations. They are marked abstract in their UML representation in Fig. 5. As a consequence, the previously mentioned factory methods are not generated after having included each feature module, but rather for the composed, final collaboration.

3.2.2 Dynamic Class-Level Feature Binding

For class-level feature composition, the objective is to derive a class structure from the collaboration classes which forms the configured product (Graph and weighted for a weighted graph). In NX, this class structure can be built on the fly, by generating a metaclass based on the product line assets. In order to build a graph product named G1 with weighted edge support from the GPL, we need the base collaboration Graph and the feature module weighted (Fig. 6).

In this scenario, the composition artifact G1 is again a collaboration class with two nested classes G1::Edge and G1::Node. This class structure represents the derived «Product» which, like any other class, can be instantiated and subclassed. Since the result of the asset transformation is a freshly configured class, the constructor of its metaclass is the natural place for performing the transformation. The input to this generative step are the base collaboration and the respective feature modules. We add these two properties to the definition of the Collaboration metaclass in Fig. 7a, lines 1–2. Upon creating a new class from the metaclass (Fig. 7a, lines 5–7), the constructor of the Collaboration metaclass performs the following steps:

1. Compute the collaboration parts based on the base class and the configured feature modules.
2. Compute the extension hierarchy for the collaboration classes and the collaboration parts.
3. Add the collaboration classes of the feature modules as superclasses of the base class.
4. Create additional nested classes in this collaboration class, one for each collaboration part. Then, these part classes are combined using multiple inheritance according to the computed extension hierarchy.
5. Add factory methods as instance methods of the generated class for creating instances of the collaboration parts on demand.

The last step provides composition locality (e.g., by returning instances of G1::Node rather than Graph::Node).

Figure 7: The Generated Collaboration Class G1

The result of composing the base Graph and the feature module weighted is shown in Fig. 6. The resulting class-level product G1 can be instantiated (see Fig. 7b; see also the last transformation in Fig. 6). The dispatch upon the print method (see line 2) proceeds from G1 to weighted and then Graph. By leveraging the built-in NX object and class generation mechanism, client components of the class-level product can use it as an ordinary class. The collaboration class (G1) can be refined further either by providing methods to it (line 1 Fig. 7c) or by subclassing (lines 3–5). The same holds for the collaboration parts (G1::Edge, G1::Node). NX’s built-in object system introspection is used during the above transformation steps to query the child objects of the collaboration classes and to extract their object names.
3.2.3 Dynamic Object-Level Feature Binding

In the second dynamic binding scenario, the feature composition is performed at the object level using the common code assets (see Section 2.1). At the object level, an instance of a collaboration class and its child instances, representing the collaborating parts, are the binding targets.

As an example, we refer to the collaboration class `Graph` and a feature module `weighted` to form a weighted graph product (see Fig. 8). In an object-level composition, one can specify the feature composition either at the time of object construction (called dynamic feature binding in [20]) or at a later time during the object’s life span. Similarly, we can remove feature modules from the graph at later times.

The refined graph instantiation `g1` is the product representation of a weighted graph (as identified by the `<Product>` tag in Fig. 8). Since the factory method of `weighted::Edge` is mixed into the object `g1`, `new edge()` calls on object `g1` to accept the additional `weight` argument (line 4, Fig. 9b). The `print` method provided by the `weighted` feature is resolved (line 5, Fig. 9b).

Since NX provides language support for decorator mixins, adding decorators does not require any kind of code refactoring or the generation of intermediate code structures (such as the decorator generator in [20]). The NX decorator mixins preserve the self-context throughout the composed collaboration, thus avoiding issues pertaining to decomposition mismatches (see Section 2.2).

As already stated, the running GPL example only depicts the most basic binding scenario, with a single feature module being included. Also, there are no class inheritance relations between the collaboration parts to be preserved by the extension hierarchy. However, NX supports the construction of complex decorator mixin chains and the decorator mixins can form their own inheritance structures to allow for incremental mixin implementations [25]. As a result, multiple feature modules (and the underlying `<mixin>` relations) can be added and deleted (lines 7 and 9, Fig. 9b) to support feature binding and feature unbinding (see Section 2.2).

3.2.4 Static Feature Binding

Under static feature binding, feature implementations are included into an application before load time, typically by a source-code generator or a specialized compiler frontend. This definition targets especially at languages which provide binding times prior to the actual runtime (e.g., compile time). Transferring the notion to a dynamic languages reveals two properties of static binding (see also Section 2.1): (a) Generating a tailored source code representation of a valid product (e.g., of the readily composed `G1`) and (b) disallowing product reconfigurations (i.e., the product code structure is fixed). The latter property is commonly motivated by baking code-level optimization (for a particular resource constrained platform) into a product and by avoiding the time penalties of dynamic feature binding [19].

Dynamic and reflective languages such as NX can meet property (a) by serializing [18] a given product of the SPL. NX provides a flexible serializer infrastructure capable of streaming objects and classes into source code, reflecting their current configuration state. Therefore, we can serialize the object-level and class-level products with no effort:

```
1 package require nx::serializer
2 foreach fm [FeatureModule info instances] {
3   puts [fm serialize]
4 }
5 puts [G1 serialize]
6 puts [g1 serialize]
```
The above snippet showcases the loading of the NX serialization and its instrumentation to create a script from the SPL instances as specified in the previous two sections. Serialization is supported both for the class-level and for the object-level compositions (see lines 6 and 7 above).

From property (b) it follows that the composed product with its refinement relations must not be changeable. Likewise, the feature composition should not be extensible (by adding further, previously omitted features). In dynamic and scripting environments as NX, feature composition is inherently subjected to change. In NX, for example, it would be possible to redefine the product structure and product behavior after restoring the product (G1) from its serialized state through reflective operations (such as altering class relations, adding new methods, redefining objects and classes). Evaluating techniques for freezing products at the object and at the class levels are future work (e.g., variants of superimposition based on runtime structures, applying filters to the serialization process).

### 3.2.5 Implementation

The full NX implementation study is given in the Appendix to this paper. The implementation, while not feature complete, is lightweight. The concepts of collaborations and feature modules map to the two metaclasses Collaboration and FeatureModule. The weaving behavior defined by these metaclasses is implemented by a small code fragment. The code necessary for computing the extension hierarchy fits in 29 SLOC, the code for feature weaving at the class level in 19 SLOC, and its object-level counterpart for weaving at the object level in 12 SLOC. This is completed by another 15 lines for adding some syntactic sugar (e.g., the infrastructure for the managing properties such as features). Despite the limitations of SLOC, this weak approximation of code size indicates the low effort required for a basic feature binding framework in NX.

### 4. RELATED WORK

Compositional approaches [20, 19, 21, 2, 24, 23, 6, 16] to feature-oriented programming (FOP) of software product lines (SPLs) typically support one or several feature binding strategies as defined in Section 2.1. Below, we review the ones which directly influenced our approach. A more complete account on binding support in FOP is given in [19].

Rosenmüller et al. [20, 19] propose code generation from a single asset base integrating both static class-level and dynamic object-level feature bindings in FeatureC++. The framework allows for switching between static class-level and dynamic object-level feature bindings at SPL build time. These assets (class refinements) are organized in a flat folder structure. For static binding, these class structures are merged by superimposition. For dynamic binding, feature classes as part of a GoF Decorator pattern idiom are generated. These feature classes are then organized in decorator chains to implement layered designs, based on method forwarding, using an application-level super-reference list. This entails decomposition issues such as the self-problem. Limitations are due to the host language C++ (e.g., not supporting dynamic class-level composition).

Ostermann [16] puts forth a collaboration-based and layered implementation technique based on prototype delegation (to realize refinement chains) and a variant of virtual classes (to represent collaborations with composition locality; see Section 2.2). The result allows for dynamic, object-level compositions. Multiple binding schemes are not supported. This delegation layer compares with our dynamic, object-level technique using NX decorator mixins. For example, decorator mixins share the rebinding of the self-reference under delegation.

Smaragdakis and Batory [24, 23] present an implementation technique for collaboration-based designs using mixin layers. Their notions of collaboration-based design and of coarse-grain modularization for step-wise refinements is also a central motivation for FOP in NX. As for the implementation techniques for collaboration-based designs and the notion of mixins, besides C++, Smaragdakis and Batory [24] explore the use of CLOS mixins (i.e., the CLOS variant of multiple class-based inheritance with linearization). Their CLOS implementation study compares with our NX study as NX’s OS system is closely modeled after CLOS (e.g., the linearization scheme used). In addition, the CLOS meta-object protocol allows for implementing versatile serializers [18] to be used as outlined in Section 3.2.4 for NX.

In CaesarJ [2] (and the Beta family of languages) the concept of family classes as collections of virtual classes attracted our attention towards the issue of composition locality and influenced the NX implementation of collaboration classes based on constructor generation and object nesting. Also, NX provides comparable means to navigate family classes. The NX helper command info parent allows one to access the enclosing object, similar to the pseudo variable out in gbeta. Further similarities result from CaesarJ and gbeta composing superclass hierarchies upon binding family classes (and their nested classes) to each other. The nested classes are a variant of abstract subclasses.

In DeltaJ [21] refinements are limited to the class level. A program is generated given a product configuration. We, therefore, classify DeltaJ as a static, class-level approach only. While in our dynamically typed language setting, we stress software compositional issues, Schäfer et al. investigate (static) type safety under feature composition.

In [6], Gunther and Sunkle introduce the Ruby FOP extension rbFeatures. The FOP approach is mainly annotation-based, that is, feature-specific code is grouped using Ruby blocks (e.g., inside a Proc object) and feature composition is then performed by evaluating an assembled set of such blocks. In our scheme in Fig. 2, this constitutes a composition level distinct from objects and classes which also covers the sub-method level, for example. This meta-programming scheme for script-level composition is implementable in NX. The authors of rbFeatures, however, do not consider object-compositional facilities, in particular Ruby modules. Although missing metaclasses, the techniques introduced in Section 3.2 (with object aggregation) can be approximated using Ruby modules and open class declarations.

### 5. SUMMARY AND CONCLUSIONS

We presented an approach to dynamic and to static feature bindings, both at the object level and at the class level. The assets of the SPL (the base collaboration and the feature modules) are represented as objects and classes, with collaboration structure being modeled through dynamic object aggregations. The same set of assets is used as the source for dynamic and static feature binding. For the implementation of the approach, we use high-level object-oriented concepts such as multiple class-based inheritance, decorator mixins,
metaclasses, object/class aggregation, and object system introspection. The resulting implementation study meets critical requirements, such as providing for a single code base, composition locality, and the avoidance of typical decomposition mismatches in collaboration-based designs. Given appropriate language support as in NX, the approach turns into a lightweight implementation (see the Appendix).

The approach presented is not complete. We have not addressed checking of product line models, evaluating composition constraints (unlike [19]), and handling feature interactions. Also, support for homogeneous and dynamically crosscutting features has not been considered. For the latter, NX provides message-level filters [13]. NX also supports conditional mixins for fine-grained composition control, based on guarding expressions. There are also mixin variants [25] available to enforce strict feature ordering. Besides, while the GPL helps demonstrate similarities and differences to prior work [20, 16], the framework’s fit regarding larger-scale SPLs remains to be evaluated.

In a next step, we will extend our feature binding framework beyond structure-preserving binding techniques to support flattening layered collaboration structures (see the merge operator in [15] and traits [4]). This is important to offer optimizations (e.g., minimizing a product’s memory footprint) under both the static and the dynamic binding modes, as well as to fully support static feature binding.

6. REFERENCES


Appendix: The NX Implementation Study

1. Feature Framework Classes
2. # The collaboration metaclass takes a base class and a set of features modules
3. # to build a new class in its constructor.
4. #
5. ns::class create collaboration -superclass ::nx::Class {
6.    :property (features=::FeatureModule *)
7.    :property (partial=switch true)
8.    -context -featureModules -baseClass::public
9.    \}
10. # Create an
11. # A FeatureModule is a specialized collaboration.
12. # Let the product inherit from the base class.
13. foreach name $featureClass \[
14.    \# Create child classes as collaboration parts.
15.    ns::class create $featureClass { -context :superclass :public method "new [string lowerName]" args \[
16.         \} \]
17. \}
18. \}
19. \]
20. \]
21. public method init () { \[
22.    \[
23.    \]
24. \}
25. \]
26. \]
27. \]
28. \}
29. \]
30. \]
31. \]
32. \]
33. \]
34. \]
35. \]
36. \]
37. \]
38. \]
39. \]
40. \]
41. \]
42. \]
43. \]
44. \]
45. \]
46. \]
47. \]
48. \]
49. \]
50. \]
51. \]
52. \]
53. \]
54. \]
55. ns::class create FeatureModule -superclass Collaboration { \[
56.    :property (features=::FeatureModule *)
57.    :property (partial=switch true)
58.    -context -baseClass::public
59. \]
60. \]
61. \]
62. \]
63. \]
64. \]
65. \]
66. \]
67. \]
68. \]
69. \]
70. \]
71. \]
72. \]
73. \]
74. \]
75. \]
76. \]
77. \]
78. \]
79. \]
80. \]
81. \]
82. \]
83. \]
84. \]
85. \]
86. \]
87. \]
88. \]
89. \]
90. \]
91. \]
92. \]
93. \]
94. \]
95. \]
96. \]
97. \]
98. \]
99. \]
100. \]
Safe Adaptation in Context-Aware Feature Models

Fabiana G. Marinho, 
Department of Computer Science 
Federal University of Ceará 
Fortaleza, Brazil 
fabiana@great.ufc.br

Rossana M. C. Andrade 
Department of Computer Science 
Federal University of Ceará 
Fortaleza, Brazil 
rossana@ufc.br

Paulo H. M. Maia 
Group of Computer Network, Software Engineering and Systems - GREat 
Federal University of Ceará 
Fortaleza, Brazil 
paulomaia@great.ufc.br

Vanina M. P. Vidal 
Department of Computer Science 
Federal University of Ceará 
Fortaleza, Brazil 
vvidal@ufc.br

Paulo A. S. Costa 
Department of Computer Science 
Federal University of Ceará 
Fortaleza, Brazil 
alexandre@great.ufc.br

Claudia Werner 
Systems Engineering and Computer Science Program 
Federal University of Rio de Janeiro 
Rio de Janeiro, Brazil 
werner@cos.ufrj.br

ABSTRACT

Software product lines, usually described using feature models, have proven to be a feasible solution to develop mobile and context-aware applications. These applications use context information to provide services and data for their users from anywhere and at any time. However, building feature models for mobile and context-aware software product lines demands advanced skills of software engineers, since it comprises system and context information. Moreover, to guarantee a correct application execution, these models must be thoroughly specified, composed and verified to check whether some composition and adaptation rules are violated. Although this is an important task, there is a lack of formalization of such rules, which makes it difficult to use those rules for feature models verification. In this paper, we propose an approach to prevent defects in context-aware feature models and in their product reconfiguration based on formal methods. To validate our work, we developed a prototype to check the correctness of context-aware feature models.

Categories and Subject Descriptors
D. SOFTWARE [D.2. SOFTWARE ENGINEERING]: D.2.4 Software/Program Verification—Assertion checkers, Correctness proofs, Formal methods

General Terms
Verification

Keywords
Context-Aware SPL, Feature Model, Formal Method

1. INTRODUCTION

Software Product Line Engineering (SPLE) is a reuse-driven development paradigm which heavily relies on domain analysis to identify variabilities to manage the differences between products. To accomplish this, SPL approaches use, in many cases, Feature Models (FMs) [18], which describe a domain by representing common and variable features of an SPL.

Technological advances in mobile devices are fostering the creation of highly distributed and interactive applications, characterized by the dynamicity and uncertainty of resources. In these applications, requirements such as mobility and context-awareness demand interoperable, uncoupled, adaptable, and autonomous programming abstractions [21]. Thus, at runtime, the environment, user requirements and interfaces between software and hardware may change dynamically, requiring a prompt response to these changes [17].

SPLs have shown to be an efficient way to handle requirements from mobile and context-awareness domain, as can be found in [17], [19], and [23]. In that direction, an SPL to support the development of context-aware applications, called Context-Aware Software Product Line (CASPL), should represent in the FM the context information relevant to the domain and describe the impact of this context information on the product adaptation.

Throughout this paper, FMs for CASPLs are called Context-Aware Feature Models (CAFMs) and are composed by two models: a System Model (SM), which expresses variabilities and similarities between features of the modeled domain, and a Context Model (CM), which represents context entities of that domain. These models are enriched by Composition Rules (CR) and Adaptation Rules (AR). The former specifies constraints among SM elements only, while the latter defines relationships among elements of the two models, such as which context information in a CM can cause an adaptation in a SM. Here, well-formedness is understood as the conformance of model elements with constraints of the underlying formal specification. Observe
that well-formedness and consistency go beyond syntactical correctness, as they also take into account semantic constraints. Ensuring that all participating models and rules in a CAFM are well-formed and consistent is necessary, but not sufficient to ensure a safe adaptation, since adaptation problems can occur only to a particular reconfiguration created at runtime. Considering this scenario, to guarantee those properties, the models that comprise a CAFM should be composed to verify the lack of defects, which represent violations of the specified rules. However, analyzing these potential defects is a challenging task, since ensuring that relevant properties and constraints are preserved during composition is essential. Therefore, a verification mechanism should be proposed to check the presence of defects that can emerge from SM and CM composition.

The main contribution of this paper is to propose an approach that aims at minimizing the presence of defects in product adaptation by predicting, at development time, the defects that may arise in a CAFM. This approach uses a specification that formalize CAFM elements and properties. To validate the proposed approach, we have developed a prototype with which the user can model CAFMs that will be automatically and transparently translated into an internal formal specification. The tool uses this new model representation to verify its well-formedness and consistency against a set of predefined properties. Therefore, the user does not need to know and deal with the formal specification.

The remainder of this paper is divided as follows. In Section 2, we discuss work related to the proposed approach. In Section 3, we present and formalize the concepts related to CAFM. In Section 4, we present the properties that correspond to the set of formal requirements for rules in a CAFM. In Section 5, we present our approach. In Section 6, we use a prototype to validate the proposed approach, and finally, in Section 7, we present our conclusions and future directions.

2. RELATED WORK

We structure our discussion of related work into three categories: analysis of FMs, modeling and well-formedness checking of CAFMs, and proposals to maximize integrity of model composition. Regarding the former category, Zaid et al. [28] and Wang et al. [27] propose an ontology in order to formalize FMs to check model consistency and conflict detection through predefined rules. Czarnecki and Pietroszek used Object Constraint Language (OCL) to validate constraint rules in [10]. Sun et al. [25] propose the use of Alloy to formalize FMs and the Alloy Analyzer tool to check FMs consistency. Gheyi et al. [16] also adopt Alloy and Alloy Analyzer to propose a generic formalization and consistency checking to FMs.

Other approaches check the consistency of FMs based on rigorous mathematical theories. For example, Zhang et al. [29], Mannion [22] and Batory [4] propose FM translation into propositional formulas. Czarnecki and Wasowski [11] propose the extraction of FMs from propositional formulas. The use of constraint programming is investigated by Benavides [5] and Trinidad et al. [1]. According to Benavides [5], the main disadvantage in those proposals is the low level of abstraction used, since they are only appropriated when FMs are analyzed using the specific formalisms and tools to each proposal. Moreover, the support for the analysis of extended FMs and CAFMs is flawed in most of the aforementioned studies.

In the second category, there are research work in the literature that address context modeling and correctness checking during variability modeling. For example, Fernandes et al. [15] propose a notation for variability modeling for context-aware SPLs, called UBIFEX-Notation. The authors also propose the UBIFEX-Simulation to minimize defects in the configured CAFM. However, there is no formalism used in that work. Ubayashi et al. [26] also propose a method for variability modeling for context-aware SPLs that treats context as a separate SPL. The authors use formal methods to specify and check the correctness of the constructed assets, but there is not any verification mechanism to validate the configured products.

Research in the latter category includes the following. According to Acher et al. [2], FMs can be separated and composed, ensuring that relevant properties are preserved during composition. However, the authors state that they did not consider model constraints. Acher et al. [3] propose a novel slicing technique for FMs. In their work, FMs have been semantically related to propositional logic. Although the authors consider cross-tree constraints, they do not address context-awareness and its implications for product adaptation. Lopez-Herrejon and Egyed [20] present C2M V2, an ongoing project whose goal consists of applying and extending work on incremental consistency management to SPLs that are developed with compositional approaches. Their proposal includes constraints from several models, providing inter-model consistency. However, we could not find any current results of this project in the technical literature.

The approach proposed in this paper aims at providing solutions to the gaps found in those works. The main drawback of the above proposals is the low level of abstraction because these approaches are appropriate only when the FMs are analyzed using the formalisms and tools specific to each proposal. Another point to be emphasized is that our approach uses a formal specification built based on First Order Logic. This fact has brought benefits in terms of rigor to our approach, since it uses a mathematical notation that explicitly addresses the semantic aspects of CAFMs. With the exception of [7] and [5], it is worth mentioning that this formal specification incorporates concepts not yet formalized in the literature concerning FMs, such as cardinality, feature attribute and composition rules, as well as concepts related to context modeling, for example, adaptation rules, entities context, context information and attributes context.

3. FORMAL SPECIFICATION OF A CAFM

We have chosen the Extended FM notation [6] to represent SMs and CMs since it incorporates a richer semantics. To illustrate a CAFM, we use the Mobile and Context-Aware Visit Guide SPL, which is a result of the MobiLine Project [24].

An SM consists of a tree structure that has a unique root r representing the modeled domain and in which nodes correspond to the features and edges describe the hierarchical relationships between these features. The remaining nodes are grouped in disjoint sets that are subtrees of r, denoted by Sn. If a node n belongs to a subtree of a node n, then n′ is successor of n and n is predecessor of n′. A CM is
Table 1: Examples - predicates to formalize CAFMs

<table>
<thead>
<tr>
<th>Predicate</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>subclass-of(X,Y)</td>
<td>specifies the relationship between an antecedent feature (X) and its descendant (Y)</td>
</tr>
<tr>
<td>mandatory(X,Y)</td>
<td>specifies a mandatory relationship between an antecedent feature (Y) and its descendant feature (X)</td>
</tr>
<tr>
<td>min(X,Integer)</td>
<td>specifies the minimum cardinality of a feature (X)</td>
</tr>
<tr>
<td>max(X,Integer)</td>
<td>specifies the maximum cardinality of a feature (X)</td>
</tr>
<tr>
<td>attribute(X,Y)</td>
<td>specifies the relationship between an attribute feature (Y) and its antecedent feature (X)</td>
</tr>
<tr>
<td>present(X)</td>
<td>specifies that the feature X is present in a CAFM</td>
</tr>
</tbody>
</table>

Table 2: Predicates to represent \( S_{\text{ExchangeType}} \)

<table>
<thead>
<tr>
<th>Predicate</th>
</tr>
</thead>
<tbody>
<tr>
<td>present(MessageExchange)</td>
</tr>
<tr>
<td>present(ExchangeType)</td>
</tr>
<tr>
<td>present(Synchronous)</td>
</tr>
<tr>
<td>present(Asynchronous)</td>
</tr>
<tr>
<td>subclass-of(MessageExchange, ExchangeType)</td>
</tr>
<tr>
<td>mandatory(MessageExchange, ExchangeType)</td>
</tr>
<tr>
<td>subclass-of(ExchangeType, Synchronous)</td>
</tr>
<tr>
<td>subclass-of(ExchangeType, Asynchronous)</td>
</tr>
<tr>
<td>min(ExchangeType, 1)</td>
</tr>
<tr>
<td>max(ExchangeType, 1)</td>
</tr>
</tbody>
</table>

Table 3: Predicates to represent \( S_{\text{ExchangeType}} \)

<table>
<thead>
<tr>
<th>Predicate</th>
</tr>
</thead>
<tbody>
<tr>
<td>&lt;CR&gt; ::= &lt;antecedent&gt; → &lt;consequent&gt;</td>
</tr>
<tr>
<td>&lt;antecedent&gt; ::= &lt;expression&gt;</td>
</tr>
<tr>
<td>&lt;consequent&gt; ::= &lt;expression&gt;</td>
</tr>
<tr>
<td>&lt;expression&gt; ::= &lt;expression&gt; ∧ &lt;logic&gt; ∧ &lt;expression&gt;</td>
</tr>
<tr>
<td>&lt;f(v:t)&gt; ::= ¬ &lt;f(v:t)&gt;</td>
</tr>
<tr>
<td>&lt;r(v:t)&gt; ::= &lt;relation&gt; &lt;domain&gt;</td>
</tr>
</tbody>
</table>
| <logic> ::= \( \land \) \lor \lor \rightarrow \leftrightarrow \]  

Once variability and context have been modeled, CAs and ARs are specified using a propositional representation. Modeling expressiveness to define CAs and ARs differs considerably in the literature ranging from just include and exclude relations to advanced propositional expressions. Here, we have adopted the latter form.

**Definition 1.** [Composition Rule] A composition rule is an implication of an antecedent expression to a consequent expression, where each expression is a propositional formula over the set of features and attribute features owned by an SM. A CR uses the following BNF:

where \( f \) and \( f(v:t) \) correspond to an optional feature and an attribute feature of an SM, respectively, and \( <\text{domain}> \) corresponds to the possible value types that can be assigned to an attribute feature.

**Definition 2.** [Adaptation Rule] An adaptation rule consists of an implication of a context expression to a system expression, where each expression is a propositional formula. The context expression comprise CM terms and the system expression comprise SM terms.

An AR sets a reconfiguration of a product by means of inclusion/exclusion of features or assignment of values for attribute features using the following BNF:

Table 4: Predicates to represent \( S_{\text{ExchangeType}} \)

<table>
<thead>
<tr>
<th>Predicate</th>
</tr>
</thead>
<tbody>
<tr>
<td>&lt;AR&gt; ::= &lt;contextExpression&gt; → &lt;systemExpression&gt;</td>
</tr>
<tr>
<td>&lt;contextExpression&gt; ::= &lt;contextExpression&gt; ∧ &lt;logic&gt;</td>
</tr>
<tr>
<td>&lt;contextExpression&gt; ::= &lt;contextExpression&gt; ∧ &lt;logic&gt;</td>
</tr>
<tr>
<td>&lt;contextExpression&gt; ::= &lt;contextExpression&gt; ∧ &lt;logic&gt;</td>
</tr>
<tr>
<td>&lt;contextExpression&gt; ::= &lt;contextExpression&gt; ∧ &lt;logic&gt;</td>
</tr>
<tr>
<td>&lt;contextExpression&gt; ::= &lt;contextExpression&gt; ∧ &lt;logic&gt;</td>
</tr>
<tr>
<td>&lt;contextExpression&gt; ::= &lt;contextExpression&gt; ∧ &lt;logic&gt;</td>
</tr>
<tr>
<td>&lt;contextExpression&gt; ::= &lt;contextExpression&gt; ∧ &lt;logic&gt;</td>
</tr>
<tr>
<td>&lt;contextExpression&gt; ::= &lt;contextExpression&gt; ∧ &lt;logic&gt;</td>
</tr>
<tr>
<td>&lt;contextExpression&gt; ::= &lt;contextExpression&gt; ∧ &lt;logic&gt;</td>
</tr>
<tr>
<td>&lt;contextExpression&gt; ::= &lt;contextExpression&gt; ∧ &lt;logic&gt;</td>
</tr>
<tr>
<td>&lt;contextExpression&gt; ::= &lt;contextExpression&gt; ∧ &lt;logic&gt;</td>
</tr>
<tr>
<td>&lt;contextExpression&gt; ::= &lt;contextExpression&gt; ∧ &lt;logic&gt;</td>
</tr>
<tr>
<td>&lt;contextExpression&gt; ::= &lt;contextExpression&gt; ∧ &lt;logic&gt;</td>
</tr>
<tr>
<td>&lt;contextExpression&gt; ::= &lt;contextExpression&gt; ∧ &lt;logic&gt;</td>
</tr>
<tr>
<td>&lt;contextExpression&gt; ::= &lt;contextExpression&gt; ∧ &lt;logic&gt;</td>
</tr>
<tr>
<td>&lt;contextExpression&gt; ::= &lt;contextExpression&gt; ∧ &lt;logic&gt;</td>
</tr>
<tr>
<td>&lt;contextExpression&gt; ::= &lt;contextExpression&gt; ∧ &lt;logic&gt;</td>
</tr>
<tr>
<td>&lt;contextExpression&gt; ::= &lt;contextExpression&gt; ∧ &lt;logic&gt;</td>
</tr>
<tr>
<td>&lt;contextExpression&gt; ::= &lt;contextExpression&gt; ∧ &lt;logic&gt;</td>
</tr>
<tr>
<td>&lt;contextExpression&gt; ::= &lt;contextExpression&gt; ∧ &lt;logic&gt;</td>
</tr>
<tr>
<td>&lt;contextExpression&gt; ::= &lt;contextExpression&gt; ∧ &lt;logic&gt;</td>
</tr>
<tr>
<td>&lt;contextExpression&gt; ::= &lt;contextExpression&gt; ∧ &lt;logic&gt;</td>
</tr>
<tr>
<td>&lt;contextExpression&gt; ::= &lt;contextExpression&gt; ∧ &lt;logic&gt;</td>
</tr>
<tr>
<td>&lt;contextExpression&gt; ::= &lt;contextExpression&gt; ∧ &lt;logic&gt;</td>
</tr>
<tr>
<td>&lt;contextExpression&gt; ::= &lt;contextExpression&gt; ∧ &lt;logic&gt;</td>
</tr>
<tr>
<td>&lt;contextExpression&gt; ::= &lt;contextExpression&gt; ∧ &lt;logic&gt;</td>
</tr>
<tr>
<td>&lt;contextExpression&gt; ::= &lt;contextExpression&gt; ∧ &lt;logic&gt;</td>
</tr>
<tr>
<td>&lt;contextExpression&gt; ::= &lt;contextExpression&gt; ∧ &lt;logic&gt;</td>
</tr>
<tr>
<td>&lt;contextExpression&gt; ::= &lt;contextExpression&gt; ∧ &lt;logic&gt;</td>
</tr>
<tr>
<td>&lt;contextExpression&gt; ::= &lt;contextExpression&gt; ∧ &lt;logic&gt;</td>
</tr>
<tr>
<td>&lt;contextExpression&gt; ::= &lt;contextExpression&gt; ∧ &lt;logic&gt;</td>
</tr>
<tr>
<td>&lt;contextExpression&gt; ::= &lt;contextExpression&gt; ∧ &lt;logic&gt;</td>
</tr>
<tr>
<td>&lt;contextExpression&gt; ::= &lt;contextExpression&gt; ∧ &lt;logic&gt;</td>
</tr>
<tr>
<td>&lt;contextExpression&gt; ::= &lt;contextExpression&gt; ∧ &lt;logic&gt;</td>
</tr>
<tr>
<td>&lt;contextExpression&gt; ::= &lt;contextExpression&gt; ∧ &lt;logic&gt;</td>
</tr>
<tr>
<td>&lt;contextExpression&gt; ::= &lt;contextExpression&gt; ∧ &lt;logic&gt;</td>
</tr>
<tr>
<td>&lt;contextExpression&gt; ::= &lt;contextExpression&gt; ∧ &lt;logic&gt;</td>
</tr>
</tbody>
</table>
| <contextExpression> ::= <contextExpression> \( \land \) \lor \rightarrow \leftrightarrow \]  

The Extended \( FM \) is a graphical notation to model SPLs. It lacks a formal syntax and semantics, which hinders the reasoning of FMs [27]. Therefore, we propose a formal specification for CAFMs that is based on First-Order Logic predicates. Table 1 depicts some predicates and Table 2 presents the set of predicates that represents the subtree \( S_{\text{ExchangeType}} \) illustrated in Figure 1.

Figure 1: Part of the MobiLine SM.

Figure 2: Part of the MobiLine CM.
where fc corresponds to a feature of a CM and CE(CM), CI(CM), and CA(CM) correspond to the sets of Context Entity features, Context Information features, and Context Attributes features of a CM, respectively. In addition, f, f(t:t) and <domain> have the same semantics as defined in the CR.

Considering Figure 1 and Figure 2, an example of a CR and an AR is shown in Table 3. The composition rule CR1 states that the presence of features Service Discovery and Service Description in the model implies the presence of the feature Message Exchange in this model. The adaptation rule AR1 states that, if the available memory is low, the feature Tuple should be present in the product reconfiguration and the size allowed for the Tuple should be 80.

<table>
<thead>
<tr>
<th>Table 5: Example - Composition Rule and Adaptation Rule</th>
</tr>
</thead>
<tbody>
<tr>
<td>Rule</td>
</tr>
<tr>
<td></td>
</tr>
</tbody>
</table>

4. PROPERTIES FOR RULES

We have defined eight properties to formally verify well-formedness and consistency of CAFMs rules. The set of identified properties results from an extensive literature review relative to the construction and formal verification of FMs (e.g., [22] [4] [8] [9] [5] [1] [29] [10] [13]). Next, we present the well-formedness properties for CRs (WFCR) and for ARs (WFAR).

**Definition 3.** [Well Formed Composition Rule] A composition rule CR is well formed for a SM if it satisfies the following properties.

WFCR1 Features referenced in a CR should be either an optional feature or an attribute feature.

WFCR2 An optional feature or an attribute feature can not require itself or one of its predecessors.

WFCR3 An optional feature or an attribute feature either in an antecedent expression or in a consequent expression should be owned by the SM.

WFCR4 An optional feature or an attribute feature can not exclude itself or one of its predecessors.

**Definition 4.** [Well Formed Adaptation Rule] An adaptation rule AR is well formed for a SM and a CM if it satisfies the following properties.

WFAR1 Features referenced in the SE should be either an optional feature or an attribute feature and should be owned by the SM.

WFAR2 A feature or an attribute feature in a SE can not be quantified more than once.

WFAR3 A feature or an attribute feature in a CE should be owned by the CM.

The consistency property regards inter-rules consistency (IRC) of a CAFM. For this, CRs and ARs are combined and the outcome is checked. A consistent inter-rule composition does not have redundant or contradictory information in the same execution scenario.

**Definition 5.** [Consistent Rules] A set of rules defined for a CAFM is consistent if it satisfies the following properties.

IRC1 CRs defined for a CAFM should be consistent with each other.

IRC2 ARs defined for a CAFM should be consistent with each other.

IRC3 CRs and ARs defined for a CAFM should be consistent with each other.

**Definition 6.** [Inconsistent CAFM] A CAFM is inconsistent when at least the conjunction of one AR and one CR is inconsistent.

An inconsistent CAFM implies that there are context situations that cause incorrect product reconfiguration. Hence, we can claim that when we prevent inter-rules inconsistencies we contribute to avoid incorrect product adaptations.

5. PROPOSED APPROACH

In this work we propose an approach based on the presented formal specification aiming at ensuring, at development time, the well-formedness and consistency of CAFMs and, consequently, improving product adaptation quality.

5.1 Rule transformation

Firstly, we translate the CAFM, the CRs and the ARs specified by the user in a high level notation to the proposed predicate notation, using a model transformation script written in ETL [14]. ETL is an Eclipse programming language which can be used to interact with EMF models to perform common Model Driven Engineering tasks such as code generation, model-to-model transformation, model validation, comparison, migration, merging and refactoring. In this sense, we defined two meta models: (i) one to express CRs and ARs in a CAFM; and (ii) another one to express the predicates. Hence, we need to transform a model that conforms to the Rule Meta Model in Figure 3 into a model that conforms to the Predicate Meta Model of Figure 4. It is worth noting that our Rule Meta Model supports attribute feature. Consequently, the expressiveness power is enhanced, since attribute features enables the Software Engineer to write specific properties involving the attributes. Therefore, it requires additional verification to avoid inconsistencies. For example, two CRs assign incompatible values to the same attribute.

Figure 3 depicts meta-classes and relationships used to capture the Rule Meta Model. The meta-classes Composition Rule and Adaptation Rule have two relationships to the meta-class Expression, representing the antecedent and the consequent expressions of a CR and the context and system expressions of an AR. Furthermore, the meta-class Expression is associated to the meta-class Feature and to the meta-classes AND, OR, and NOT, meaning that CRs
and ARs are composed of expressions, which are composed of features, feature attributes, and logical operators.

The user builds propositional formulas to express CRs and ARs. Those formulas can relate or nest multiple logical operators in accordance with the user need. To enable constructing such formulas using predicates in the meta model presented in Figure 4, we have established that a predicate could also be a parameter. Therefore, the meta-class Parameter is a specialization of the meta-class Predicate. Furthermore, a Predicate can be associated with other Predicate. The tagged values have been used to represent an attribute value and rule expressions, respectively. However, CRs and ARs are defined by the user, so the transformation rules are determined by specific user needs.

5.2 Rule correctness and consistency

We use the well-formedness properties described in Definition 3 and Definition 4 to check rules correctness. For this, those properties have been translated to the predicate notation. For example, Listing 1 presents WFCR1 in predicate notation. These predicates specify a query that is applied to the rule and determines if it is correct.

Listing 1 - Properties using predicates

\[
\begin{align*}
\text{wfcr1}(X,Y) & : - \text{present}(X), \text{present}(Y), \\
& \quad \text{subclass-of}(X,Y), \text{different}(X,Y), \\
& \quad \text{optional}(X,Y), \text{attribute}(X,Y).
\end{align*}
\]

Inter-rule consistency uses the consistency properties. For this, the rules are transformed in the predicate notation. In the CRs consistency checking, the set of CRs should be combined into a conjunction and, if this conjunction evaluates true, then the set of CRs is consistent. To verify ARs consistency, we get the set of ARs, with which context expressions can be fulfilled simultaneously, and combine the corresponding system expressions into a conjunction. If this conjunction evaluates true, the set of ARs is consistent.

5.3 Anomalies identification

The prototype also checks whether the CAFM contains anomalies (false optional, dead features and wrong cardinalities). A feature is a false optional if it is present in all derived products. To check this situation, the prototype assigns the predicate not(present) to each optional feature at a time. If the resulting formula is not satisfactory, so this CAFM has a false optional feature. To check dead features the prototype assigns the predicate (present) to each optional feature at a time. If the resulting formula is not satisfactory, so this CAFM has a dead feature. Cardinality is checked only in the next phase (SM consistency checking).

5.4 System Model consistency checking

Once the CAFM does not contain dead or false optional features, we check whether this configuration is consistent. This is achieved by transforming the SM into its corresponding predicate notation that is submitted to a Prolog Engine in conjunction with the predicates relative to the CRs, and the SM well-formedness properties. If the resulting formula is evaluated true, then the CAFM is consistent, in other words, it yields at least one product derivation.

5.5 Product correctness checking

Just after the user derives an initial product, we check if this configuration is correct. This is achieved by transforming the product into its corresponding predicates notation that is submitted to a Prolog Engine in conjunction with the predicates relative to the CRs, and the SM constrains predicates. If the product satisfies those constraints, then it is correct.

5.6 Simulation process

The most naive idea would be randomly taking a range of values as large as possible and checking them against the ARs in order to see if they are triggered. This kind of approach can be time consuming and inefficient. Our approach focuses on wisely choosing the values that triggers a set of ARs. The simulation process starts subscribing a context change, based on context entities in the CM. The subset of ARs that have been activated due to the simulated context values is created. Following that, a conjunction of the predicates relative to the actions of the activated ARs is generated. Next, the predicates corresponding to the initial product P, the CRs and the conjunction of the actions are merged. If this merge is evaluated true, we have a safe adaptation; otherwise, we have an unsafe adaptation and no change in the current product is performed. The simulation process proceeds subscribing other context changes until the previously established limit of context changes is reached (max). This limit is defined by the Software Engineering.

To ensure the greatest possible number of combinations (greatest scope) an meta-heuristic algorithm was conceived. Since we are interested in satisfying the events of each adaptation rule in order to see which of them are triggered simultaneously, our algorithm focus on generate specific values to the atomic formulas of adaptation rules. To do it, we create predicates to each of these atomic formulas. Once it is done, we can achieve all possible combinations by submitting them to Prolog. However, some atomic formulas can reference the same attribute feature. When this happens, it is mandatory to guarantee that the predicates originated from atomic formulas referencing the same attribute feature are not conflicting. In order to do it, we add some rules that specify restrictions among predicates must be respected. Finally, predicates and restrictions between them are submitted to a Prolog Engine that will identify every
possible way to satisfy the rules that will be send each at a time to the simulation process, which will check activated adaptation rules. As aforementioned, this is repeated until a max number is reached.

Therefore, we can claim that our simulation process can predict, at development time, incorrect adaptations of context-aware products. However, to ensure a complete check of product adaptation implies that the proposed approach is performed for all scenarios that lead to an adaptation. Ensuring the verification of all possible scenarios is a complex task, as the number of possible adaptations can grow exponentially and there is a great probability of a scenario that has not been foreseen occur during the execution of a product.

6. VALIDATION OF THE PROPOSED APPROACH

To validate the proposed approach, we implemented an Eclipse-based prototype and choose the Prolog language to analyse well-formedness and consistency. Prolog was chosen, since it enables to write logical specifications of searches and get them executed without recoding into another language. Furthermore, it permits to read Prolog expressions from a file and then execute it, or build it on the fly and then execute it. That flexibility is very useful, as it can provide an automatic and transparent formal verification.

To initiate the validation, rules and models should be specified using the prototype. For this, a high level interface is provided, which uses a UML Meta Model that describes the elements of a CAFM. In this work, models and rules have a graphical representation in a tree-like structure. Figure 5 and Figure 6 show an SM and a CM represented using the prototype interface. This UML Meta Model allows the use of OCL verification. The prototype guides the user in the process of model/rule construction and product configuration, hence the inclusion of defects is minimized.

Figure 5: Mobiline - Part of the SM using the prototype.

In Figure 5, the SM is composed by a root feature, which comprises the features: o2, Variation v1 (OR), m1 (mandatory), and o1. Variation v1 has four features (o3, o4, o5, and o6) as variants. Feature m1 is mandatory and is implemented by the attribute feature Attribute attr2 and by feature m2 that has an attribute Attribute attr1. Finally, feature o1 is composed by feature o7 and feature o8. Feature o7, in turn, is implemented by the feature o9. In Figure 6, the CM is composed by a Context root, which has four context entities: Context Entity ent1, Context Entity ent2, Context Entity ent3, and Context Entity ent4, and each context entity has one context information.

Figure 6: Mobiline - Part of the CM using the prototype.

CRs and ARs are modeled as logical implications and also have a graphical representation in a tree-like structure. Figure 7 and Figure 8 show examples of a CR and an AR, respectively. CR2 states that if o4 is absent and o6 is present, then m1:attr2 > 25 and m1:attr2 < 50 should be ensured. AR1 states that if ent1:inf1 > 5 and ent2:inf2 = 20, then o3 will be inserted in the current product and o6 will be removed.

Figure 7: Mobiline - Building a CR using the prototype.

Figure 8: Mobiline - Building an AR using the prototype.

Once the models and rules are specified, the prototype invokes a transformation script to generate the respective predicates. Next, the prototype evaluates CRs consistency. In this case, CRs = CR2, which is satisfiable. Then the prototype verifies ARs consistency building a set of ARs, with which context expressions can be fulfilled simultaneously and combine the corresponding system expressions into a conjunction. In this example, ARs = AR1, so the system expressions conjunction SE = (o3 ∧ ¬o6) that evaluates true, then the set of ARs is consistent.

Following that, the prototype checks the presence of false optional features. For this, the predicate not(present) is assigned to each optional feature at a time in the SM. In our example, the resulting formula is unsatisfiable, so this SM
has one false optional feature. On the other hand, assigning the predicate \( \text{present} \) to each optional feature at a time and the \( SM \) is satisfiable, then this \( SM \) does not have dead features.

To check if the \( SM \) is consistent, the prototype evaluates the conjunction of the predicates corresponding to the \( SM \) with the predicates relative to the \( CRs \), and predicates of the well formedness-rules. In this case, the \( SM \) is consistent.

Next, the user configures a product and the prototype checks whether this product is correct. For this, the prototype transforms the product configuration into predicates that are submitted to the Prolog Engine in conjunction with the predicates relative to the \( CRs \) and the \( SM \) constraints. If the resulting formula is evaluated true, then the current product is correct.

Finally, the prototype uses the simulation process, which subscribes event changes to context and gets the \( ARs \) that have been activated. Next, it evaluates if the \( AR1 \) violates the \( CR2 \), implying that the \( SM \) derives an unsafe adaptation. First, it verifies if any inconsistency between the formulas that comprise the context expressions \( CE \) exists. In this case, \( CE \) evaluates to true. Next, the prototype creates the conjunction of the system expressions \( SE \) relative to the \( CE \). In this example, it verifies \( (SE \) and \( CR2 \) \) and finds the following inconsistency \( \neg \text{present}(o6) \) and \( \text{present}(o6) \). So, \( AR1 \) violates the \( CR2 \) and the \( CAFM \) can generate unsafe adaptations.

In summary, if the actions relative to \( ARs \) break one or more \( CR \), the \( CAFM \) has inconsistencies and derives unsafe adaptations regardless the product configuration. However, there are situations in which the actions break one or more \( CRs \) just in a specific adaptation. To detect this problem, the simulation process is essential. For example, let \( CR^\prime = \{ \text{Storage is present and Wifi is absent} \} \) then \( \text{Record Movies is enabled} \) and \( SE = \text{Include Storage} \). To determine if this \( SE \) breaks \( CR^\prime \) it is necessary to check the actions against the current product configuration, since the feature \( \text{Include Storage} \) is not present in \( CR^\prime \). For instance, if the current product configuration does not contain the feature \( \text{Record Movies} \), then the configuration breaks the \( CR^\prime \), otherwise, the actions in the subset, when applied over the configuration, do not generate an unsafe adaptation. Accordingly, we can state that well-formedness and consistency properties are necessary to identify unsafe adaptations, but they are not sufficient.

7. CONCLUSIONS AND FUTURE WORK

In this work, we have proposed an approach that aims at preventing, at development time, defects of product adaptation in CASPLs. This approach comprises well-formedness and consistency verification of rules specified in CAFMs, and preventing defects in product adaptation. We used a formal specification to capture the models and rules used to build CAFMs. This specification was enriched with properties that define how to build well-formed and consistent CAFMs rules. This approach is validated by a prototype that automates the proposed verification process. The prototype uses model transformations to automatically generate the respective predicates to CAFM, the composition rules and adaptation rules and invokes a theorem prover to formally verify the proposed properties against those elements. So, the formal verification is performed in a transparent way to the end user. The scalability is directly related to the solver used. In our case, to run queries in Prolog, the selected tool limits the scalability of the proposed approach.

The simulation process proposed succeeded to detect and prevent defects in products adapted due to changes in the context. The fact that this simulation process ensure the correctness of a variety of situations that can lead to reconfigurations of products can also be considered an important outcome of this research. In order to use the proposed simulation process at run time, it would be necessary to use some framework, for instance the WildCat Toolkit [12], that automatically detects the presence of new context entities. In addition, the user must set the new scenarios that have be monitored at every new entity identified.

As future work, we intend to apply our approach in other CASPLs, such as HSR Product Line [19], in order to analyze better its benefits. Unfortunately, we did not find many CASPLs in the literature. Hence, we intend to perform an experiment using randomly generated CAFMs. Additionally, self-adaptation, maintenance, and evolution of SPL are topics of increasing interest to the SPL community. We believe the combination of the proposed approach and self-adaptive approaches could bear a significant step to predict the quality of product adaptation, since no significant change is necessary. For this, it is necessary to use some mechanism that some mechanism that has the ability to recognize new context entities and insert them in the adaptation rules. User participation is necessary to define new adaptation rules based on the new context entities identified. Thus, our work can potentially contribute to the ongoing research on those topics.

8. ACKNOWLEDGMENTS

This work is a partial result of the UbiStructure project supported by CNPq (MCT/CNPq 14/2011 - Universal) under grant number 481417/2011-7. Thanks also to CNPq for the scholarships of Rossana M.C. Andrade under grant number 314021/2009-4, Fabiana G. Marinho under grant number 552924/2008-3, and Claudia Werner under grant number 304097/2010-1.

9. REFERENCES


[22] M. Mannion.


Towards a Catalog of Variability Evolution Patterns: The Linux Kernel Case

Leonardo Passos  
University of Waterloo  
lpassos@gsd.uwaterloo.ca

Krzysztof Czarnecki  
University of Waterloo  
kczarne@gsd.uwaterloo.ca

Andrzej Wąsowski  
IT University of Copenhagen  
wasowski@itu.dk

ABSTRACT

A complete understanding of evolution of variability requires analysis over all project spaces that contain it: source code, build system and the variability model. Aiming at better understanding of how complex variant-rich software evolve, we set to study one, the Linux kernel, in detail. We qualitatively analyze a number of evolution steps in the kernel history and present our findings as a preliminary sample of a catalog of evolution patterns. Our patterns focus on how the variability evolves when features are removed from the variability model, but are kept as part of the software. The identified patterns relate changes to the variability model, the build system, and implementation code. Despite preliminary, they already indicate evolution steps that have not been captured by prior studies, both empirical and theoretical.

Categories and Subject Descriptors

D.2.7 [Distribution, Maintenance, and Enhancement]: Restructuring, reverse engineering, and reengineering

General Terms

Design

Keywords

variability, patterns, evolution, software product lines, Linux

1. INTRODUCTION

Variability evolution is a core point in evolving software product lines [6]. Changes in the variability dictate which features are obsolete, which are new, which products are still possible to be generated, which dependencies still hold, etc. Despite its importance, the Software Product Line community has little knowledge on how variability evolution occurs in practice and which changes are performed when realizing them. The few existing studies do not take feature removal into account [4, 5, 12], while others [14, 8] focus on the variability model alone. Altogether, they fail to cover the variability evolution when features are removed from the variability model, while still being kept part of the software. To address this issue, we study a real world variant rich software—the Linux kernel—and extract evolution patterns describing how variability evolves across different artifacts (variability model, build files, and source code) when features are erased from the variability model, but not from the software itself.

The Linux kernel is the most successful open source software, containing a rich and extensive variability that allows it to support a large range of architectures, device drivers and application domains [15].

Variability in the Linux kernel is vertically present in three separate, but related spaces [10]: configuration space: kernel configuration files (Kconfig), comprising the Linux variability model; compilation space: kernel build files (KBuild), mostly written as Makefiles with implicit rules [16]; implementation space: realization of all features, mostly written as C code.

The Linux kernel configuration space was first studied by She et al. [14], who analyze and compare its complexity with regards to existing models in SPLoT [9]. Lotufo et al. [8] extend that work by a longitudinal analysis over the x86 architecture. Among other things, the authors inspect the Linux variability model growth pace, how its structure is affected and which changes developers execute over time.

A sole focus on the configuration space, however, does not provide a full understanding of how variability evolves. In fact, such an analysis can easily lead to wrong conclusions. The variability model of the x86_64 architecture illustrates that: between releases 2.6.32 and 2.6.33, 281 new feature names were added, while 43 were removed. A closer inspection of all spaces of the commits removing such features led us to conclude that 35% of them continued to exist; as our patterns show, developers remove these features from the variability model while migrating them to the implementation side or merging them with other features.¹

The patterns we present is the first work capturing variability evolution in a multi-space setting of a complex real-world variant rich software. Furthermore, our patterns comprise evolution steps not covered by previous work [4, 5, 12, 14, 8].

We believe that a holistic understanding of evolution practice of complex systems with rich variability will have significant impact on product line research, including work

¹ Renames were also noted.
on methodologies, architectures, modeling languages, automatic analyses and tooling.

The rest of this paper is organized as follows: in Sec. 2 we provide a comprehensive understanding of the three spaces of the Linux kernel, and how they relate to each other. In Sec. 3 we discuss the methodology for extracting our catalog of evolution patterns, which are then presented in Sec. 4. In that section, we show the structure of each pattern, with concrete examples and discussion. We then analyze possible threats to validity of our findings in Sec. 5, and present related work in Sec. 6. We conclude the paper in Sec. 7, along with directions for future work.

2. BACKGROUND

The variability in the Linux kernel appears in three main spaces: (i) configuration space, comprised of Kconfig files; (ii) compilation space: set of kernel build files (KBuild), and; (iii) implementation space: mostly C source code. We present them now in more detail.

Configuration space.

Kconfig is the language in which features and their dependencies are declared. The kernel configurator (xconfig) renders the Kconfig model as a tree of features, from which users select the ones of interest (see Fig. 1). For instance, users interested in a cluster file system can select the OCSFS2 (Oracle™ Cluster File System) feature, whose Kconfig snippet is shown in Fig. 2.

Features in Kconfig are mostly written as configs (Fig. 2, lines 3 and 12), and may contain attributes such as type, prompt, dependencies, implied selections, default values, and help text. In our example, OCSFS2 is a tristate feature (line 4): it can be absent (n) or users can select it to be either compiled as a dynamically loadable module (m – shown as a dot in Fig. 1) or statically compiled into the resulting kernel (y – shown as a tick in Fig. 1). Boolean features (line 13) are also possible, assuming either y or n as value. Other types include integer and strings (not shown). A prompt message is a short description of a feature (lines 4 and 13), and it is used by the configurator when rendering the feature in the hierarchy. Features without a prompt are not visible to users. Dependencies (line 5) state a condition that must be satisfied to allow selection of the feature. A select attribute (line 6) enforces immediate selection of target features (CONFIGFS_FS). A default attribute (line 15) states the initial value of a feature, which might later be changed in the configuration process. The feature hierarchy depends on the order in which features are declared and on their dependencies. Cross tree constraints are defined using select and depends on attributes, but also by default values in combination with visibility conditions. Visibility conditions and default conditions (not shown) are guard expressions over feature names that follow prompt and default attributes: but also by default values in combination with visibility conditions. Visibility conditions and default conditions (not shown) are guard expressions over feature names that follow prompt and default attributes: for prompts, it controls whether the feature should be made visible; for defaults, it controls which default attribute is applicable when more than one is defined. For a full mapping from Kconfig to standard PODA feature models, refer to [14, 3]. Formal semantics of Kconfig is presented in [13].

The configurator generates a .config file, which is basically a sequence of (feature-name, feature-value) pairs. Given the

![Figure 1: Linux configurator (xconfig)](image)

features OCSFS2_FS (OCSFS file system support) and OCSFS2_FS_POSIX_ACL (OCSFS POSIX Access Control Lists) as configured in Fig. 1 results in the following .config snippet:

```
CONFIG_OCSFS2_FS=m
CONFIG_OCSFS2_FS_POSIX_ACL=y
```

Compilation space.

The KBuild system controls the compilation process of the Linux kernel. In KBuild, the files containing compilation rules are essentially Makefiles with implicit rules [16]. The image of the kernel is defined by the vmlinux-all goal contained in a top Makefile, whose snippet is shown in the first part of Fig. 3. To build the image, vmlinux-all requires the object files of the symbols appearing at the right hand side of the goal (line 3), which are then linked together. In that case, it requires all the object file names stored in core-y, libs-y, drivers-y and net-y variables. These variables denote lists of object files to which other elements can be appended to. If directories are appended (line 5), KBuild recursively runs the Makefile contained in each such directory and generates one object file per directory based on the content of a special list: obj-y (similarly, a list obj-m controls module compilation). Objects may be conditionally added to this list by replacing y with a feature name. As shown in the second fragment of Fig. 3 (line 3), ocfs2.o is only added to obj-y if the feature OCSFS2_FS is set to be y in the .config file. KBuild attempts to compile object files by locating a corresponding C file matching the same name. However, that is not always the case. For ocfs2.o, there is no ocfs2.c file in the Makefile’s directory, so KBuild relies on a list named ocfs2-objs (line 11) as the set of object files that should compose ocfs2.o. As before, objects may be conditionally added to such a list (line 10).

Implementation space.

Variability in the source code base is expressed in terms of conditional compilation macro directives, whose conditions are Boolean expression over feature names (see Fig. 4). It is worth noting that before KBuild compiles any code, it reads the content in the .config file and creates an autoconf.h header file containing macro definitions for all features that should be part of the kernel, along with their values. KBuild forces this file to be included in all C sources (this is achieved using gcc’s -include switch). For instance, selecting OCSFS2_FS_POSIX_ACL for the OCSFS2_FS module results in a definition such as

---

63
3. METHODOLOGY

We collected four patterns from a selection of 140 among 220 feature removals from the configuration space in three kernel release pairs of the x86_64 Linux kernel: (v2.6.32, v2.6.33), (v2.6.26, v2.6.27) and (v2.6.27, v2.6.28). Each pattern documents a situation in which the feature is removed from the configuration space, but continues to exist in the software.

Three of our patterns come from the analysis of (v2.6.32, v2.6.33). Our particular interest in v2.6.32 regards to the fact that it is the baseline kernel in Debian 6.0,\(^3\) one of the most mature and popular distributions in the Linux community.

From this initial analysis, we aimed at sequentially diffing release pairs starting from v2.6.26. We fixed such starting point due to incompatibility issues when using newer kernel build infrastructure with older Kconfig and .config files.

While we analyzed and classified all 43 removals in the pair (v2.6.32, v2.6.33), the selection of removals for analysis in (v2.6.26, v2.6.27) and (v2.6.27, v2.6.28) was rather arbitrary. Our main concern was only to capture a pattern that we had not seen before.

Our infrastructure is built on top of the KBuild system, which we extracted from the Linux source code. With it, we parse Kconfig files and compute the set difference of the features in each pair of kernel releases. To facilitate analysis, we also created a relational database containing all feature additions and removals, which are linked with the associated release pair and commit identifier. The records in this database were constructed by parsing all patches in the Linux Git repository.

Our analysis is based on manual inspection over the collected set of commit patches. Since changes can span more than one commit, whenever a patch is insufficient to draw a sound conclusion, we set to recover other commits changing the feature under investigation or any other feature that may affect it (e.g.: a parent feature).

4. EVOLUTION PATTERNS

This section presents in detail four evolution patterns in commits found in the Linux kernel repository.

To reduce clutter, we present each pattern in an abstract manner, capturing the changes in each artifact type. Then, we rely on fragments of real artifacts to exemplify the presented concepts, followed by a discussion of the pattern.

We present the first pattern as a basic walk-through to our notation and adapt it as we proceed with presentation.

4.1 Optional feature to implicit mandatory

In this evolution pattern, depicted in Fig.5, an optional feature \(F\) is removed from the feature model, but becomes unconditionally compiled in source code. Its compilation, however, is subject to the presence of \(F\)'s parent \(P\).

The pattern is presented in two parts, capturing the structure before the change (shown at left) and after it (shown at right). It abstractly documents changes to a fragment of the variability model (rendered in the FODA notation), shown inside a dashed box; the build artifact \(B\); source code \(C\), and; the cross-tree constraint formulae (CTC).

\(\text{Instance.}\)

\(^3\)http://www.debian.org/
\(^4\)git://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git
With our notation, build files are denoted as a sequence of objects that the compilation condition guarding `acl.o` is dropped, which we explicitly represent by writing it as crossed:

\[ B' = <..., (P, P.o += F.o),...> \]

Figure 5: Optional feature to implicit mandatory

<table>
<thead>
<tr>
<th>Line</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td><code>diff --git a/fs/ocfs2/Kconfig b/fs/ocfs2/Kconfig</code></td>
</tr>
<tr>
<td>2</td>
<td><code>config OCFS2_FS</code></td>
</tr>
<tr>
<td>3</td>
<td><code>+ select FS_POSIX_ACL</code></td>
</tr>
<tr>
<td>4</td>
<td><code>- config OCFS2_FS_POSIX_ACL</code></td>
</tr>
<tr>
<td>5</td>
<td><code>- bool &quot;OCFS2 POSIX Access Control Lists&quot;</code></td>
</tr>
<tr>
<td>6</td>
<td><code>- depends on OCFS2_FS</code></td>
</tr>
<tr>
<td>7</td>
<td><code>- select FS_POSIX_ACL</code></td>
</tr>
<tr>
<td>8</td>
<td><code>- default n</code></td>
</tr>
<tr>
<td>9</td>
<td><code>- help</code></td>
</tr>
<tr>
<td>10</td>
<td><code>- Posix Access Control Lists (ACLs) support</code></td>
</tr>
<tr>
<td>11</td>
<td><code>- permissions for users...</code></td>
</tr>
<tr>
<td>12</td>
<td><code>diff --git a/fs/ocfs2/Makefile b/fs/ocfs2/Makefile</code></td>
</tr>
<tr>
<td>13</td>
<td><code>ocfs2-objs := ver.o</code></td>
</tr>
<tr>
<td>14</td>
<td><code>...</code></td>
</tr>
<tr>
<td>15</td>
<td><code>xattr.o</code></td>
</tr>
<tr>
<td>16</td>
<td><code>- ifdef $(CONFIG_OCFS2_FS_POSIX_ACL),y</code></td>
</tr>
<tr>
<td>17</td>
<td><code>- ocfs2-objs += acl.o</code></td>
</tr>
<tr>
<td>18</td>
<td><code>- endif</code></td>
</tr>
<tr>
<td>19</td>
<td><code>+ acl.o</code></td>
</tr>
<tr>
<td>20</td>
<td><code>diff --git a/fs/ocfs2/acl.h b/fs/ocfs2/acl.h</code></td>
</tr>
<tr>
<td>21</td>
<td><code>-#ifdef CONFIG_OCFS2_FS_POSIX_ACL</code></td>
</tr>
<tr>
<td>22</td>
<td><code>extern int ocfs2_check_acl(struct inode *, int)</code></td>
</tr>
<tr>
<td>23</td>
<td><code>extern int ocfs2_acl_chmod(struct inode *)</code></td>
</tr>
<tr>
<td>24</td>
<td><code>...</code></td>
</tr>
<tr>
<td>25</td>
<td><code>else</code></td>
</tr>
<tr>
<td>26</td>
<td><code>-#define ocfs2_check_acl NULL</code></td>
</tr>
<tr>
<td>27</td>
<td><code>-static inline int ocfs2_acl_chmod(struct inode *inode)</code></td>
</tr>
<tr>
<td>28</td>
<td><code>-( return 0; )</code></td>
</tr>
<tr>
<td>29</td>
<td><code>- ...</code></td>
</tr>
<tr>
<td>30</td>
<td><code>-endif</code></td>
</tr>
<tr>
<td>31</td>
<td><code>...</code></td>
</tr>
</tbody>
</table>

Figure 6: A patch matching the pattern in Fig. 5

The patch fragment in Fig. 6 is a concrete example of this pattern, where `OCFS2_FS` is `P` and `OCFS2_FS_POSIX_ACL` is `F`. In the patch, changes are either removal (lines prefixed with `"-"`) or addition (lines prefixed with `"+"`). Lines without any prefix are used as context to ease understanding.

The patch shows that the feature `OCFS2_FS_POSIX_ACL` is being removed from the feature model (lines 4–11), but its implied selection attribute is moved to its parent feature (line 3). Fig. 5 captures this situation by deleting `F` from the feature model and by replacing any references to `F` with `P` in the set of cross tree constraints, thus leading to a new set CTC'.

Regarding the changes in the Makefile, the patch shows that the compilation condition guarding `acl.o` is dropped (lines 17–19), and `acl.o` is unconditionally added to the list of objects `ocfs2-objs` (line 20). To capture this abstractly, we first introduce a simplified representation for build files. In our notation, build files are denoted as a sequence `B` of build rules of the form `(e, r1, r2)`, where `e` is a guard expression over feature names (as in line 17 of the patch); `r1` is a build rule in case `e` evaluates to true; and `r2` is the alternative build rule to be used in case `e` does not hold. For simplicity, the condition may be omitted (taken as true) to represent unconditional build rules. Moreover, the second rule may not be shown, stating the absence of an alternative rule in case the guard expression fails. Using this notation, we capture the change over the Makefile shown in the patch as follows: in the left side, `(F, P.o += F.o)` is one build rule in `B`, stating that if `F` is present, then `F`'s object code should be part of `P`'s.

After the change is applied, a new sequence `B'` is obtained containing a new build rule where the condition over `F` is dropped, which we explicitly represent by writing it as crossed:

\[ B' = <..., (K, P.o += F.o),...> \]

As for the edits in the source code side (see `acl.h`: lines 24–33), the patch indicates that the code guarded by a conditional compilation directive is kept, while the associated condition (line 24) and the alternative code block (lines 28–33) are removed. We capture this situation in our abstraction by removing specific parts (shown as crossed) of guarded blocks, which we represent as triples `(e, Cx, Cy)`: similar to build rules, `e` denotes a conditional macro expression over feature names, whereas `Cx` is the code to be compiled in case `e` holds; otherwise `Cy` is used.

Discussion

The purpose of this pattern is to guarantee that a security feature is not unintentionally left unselected in face of its parent feature presence; thus, it eliminates the chance of misconfigurations, with the cost of a bigger product (executable binary size). In our example, making Posix Access Control Lists a mandatory feature for the OCFS2 file system is in tune with that: in Linux, ACL controls file/directory permissions for groups and individuals, and it is a major security feature already supported by other filesystems, including ext3/4, xfs, btrfs, etc. In server environments using a cluster based filesystem, it is likely the case that such support is required, and its absence (unintentional or not) might lead to major security flaws, as no permission control would exist.

Interesting enough, users configuring new versions of the kernel in which `OCFS2_FS_POSIX_ACL` is not available as a selectable feature may conclude that OCFS2 dropped support for ACL. This occurs because the patch removing the `OCFS2_FS_POSIX_ACL` feature from Kconfig does not update the help text of `OCFS2_FS` to state that ACL is now an integral part of it; thus, users might not select OCFS2 as part of the kernel, driven by the conclusion that it now lacks a feature it once supported.

4.2 Computed attributed feature to code

In this evolution pattern, shown in Fig. 7, an invisible feature `F` (no prompt) is defined by a default expression `e`. The purpose of `F` is to be a mere value place holder that is referred in code using the feature’s name. The change removes `F` from the feature model, while replacing its usage in code by its computed default expression. The build artifacts and the set of cross tree constraints are not altered, meaning that `F` is not referred in constraints and it does not have an

5Commit id: e6aabe

6Kconfig does not allow arbitrary non-Boolean expressions
Otherwise, its associated bit receives a value.

In case the flag is not set (the conditional statement is not true), the bit-or assignment as shown has the same effect as before, but using a different implementation technique.

As can be seen, the above definition lacks a prompt message, and thus the feature is not visible to users. Its value is given by a combination of default conditions (refer to Sec. 2), and depends on the presence of CFG80211_DEFAULT_PS. These conditions denote a single abstract conditional expression

\[ \text{CFG80211_DEFAULT_PS} \ ? \ 1 \ : \ 0 \]

As can be seen, the above definition lacks a prompt message, and thus the feature is not visible to users. Its value is given by a combination of default conditions (refer to Sec. 2), and depends on the presence of CFG80211_DEFAULT_PS. These conditions denote a single abstract conditional expression

\[ \text{CFG80211_DEFAULT_PS} \ ? \ 1 \ : \ 0 \]

In the source code, the feature is originally referred by

```c
rdev->wiphy.ps_default = CONFIG_CFG80211_DEFAULT_PS_VALUE;
```

which was later changed to

```c
#undef CONFIG_CFG80211_DEFAULT_PS
rdev->wiphy.flags |= WIPHY_FLAG_PS_ON_BY_DEFAULT;
#define CONFIG_CFG80211_DEFAULT_PS
```

The inspected patch shows that a set of related Boolean flags in the source code, including `ps_default`, became a single integer variable (`flags`) implementing a bit mask. In that sense, the bit-or assignment as shown has the same effect as before, but using a different implementation technique. In case the flag is not set (the conditional statement is not compiled), the corresponding bit position defaults to zero. Otherwise, its associated bit receives 1 as value.

Discussion.

This pattern affects the set of configurations derivable from the configuration space, but it preserves behaviour in all products containing P, as our instance showed. In that sense, the pattern documents a refinement scenario. The existing theory over software product line refinement [5] fails to address this, as its theorems only cover situations with feature model equality or equivalence in the set of possible configurations (our .config files).

Contrary to the previous pattern, this evolution pattern is a refactoring, as it preserves behaviour and improves maintainability, at least as stated by developers in the commit.

8Committed: 5be83d

log message:

“We’ve accumulated a number of options for wiphys which make more sense as flags as we keep adding more. Convert the existing ones.”

The choice of having features as place holders for computed attributes in Kconfig files appears to be mere idiomatic preference, as there is no mentioning in the kernel coding style [9] and Kconfig language reference [10] stating which practice is preferable.

4.3 Merge features by module aliasing

This evolution pattern, illustrated in Fig 8, merges features F1 and F2 into the existing feature F1 when the implementation of F1 subsumes F2. The source code comprising the compilation unit of F2 is completely removed, and so is any build rule. Any constraints defined by F2 are deleted, and existing constraints remain as is, which means that F2 is not referred in any other constraint. Furthermore, F1 registers itself as an alias module to F2. In that case, whenever the kernel receives a request to load F2, F1 is the actual module that gets loaded.

Instance.

An instance of this pattern concerns the merge of the feature RT3090 (matches F1) into RT2860 (matches F2), with RT2860 supporting both Ralink 2860 and 3090 wireless chips. In the patch associated with this instance, all the code related to RT3090, its Kconfig entry and build files are removed. The only addition in the patch occurs in rt2860/pdi_main_dev.c:

```c
+ MODULE_ALIAS("rt3090sta");
```

where rt3090sta is the original object filename created for RT3090, as defined by the rt3090sta-objs list in its Makefile. In the above statement, RT2860 declares that it has RT3090 as its alias.

Discussion.

Merge by alias is only possible for features that are not scattered in code, but rather have a well defined set of files that once compiled generate a single object code.

Contrary to the instance found in Optional feature to implement.
licit mandatory, the description and help message of the RT2860 feature are updated to reflect the fact that it now supports the RT3090 family of chips.

It appears that RT2860 inherits much of the code from RT3090, suggesting co-evolution of the two drivers. Running the code clone detection tool CCFinder \cite{7} supports our claim, as we found 864 clones between the two drivers, with clones containing as many as 2,500 tokens (see Fig. 9 for the whole distribution). Curiously, RT2860 is smaller than RT3090, as we observed by running CLOC.\cite{8} Table 1 shows a reduction of ≈ 32\% in SLOC in comparison with RT3090’s (.h and .c files), with a Makefile 27\% more compact. Despite such a simplification in code, functionality has not been lost, as developers state in the commit log:

“Remove no longer needed rt3090 driver. rt2860 handles now all rt2860/rt3090 chipsets.”

In Linux, it is possible to create a single driver supporting multiple devices. This mechanism is also used by developers as a means to merge features. For instance, the driver for the light sensor device TSL2561 is now merged into TSL2563,\cite{9} which supports four devices, as declared in its device table:

```
static const struct i2c_device_id tsl2563_id[] = {
   { "tsl2560", 0 },
   { "tsl2561", 1 },
   { "tsl2562", 2 },
   { "tsl2563", 3 },
};
MODULE_DEVICE_TABLE(i2c, tsl2563_id);
```

Structurally, this instance is very much related to the instance previously discussed. Its difference relies on how these two features evolved: TSL2563 was implemented completely separate from TSL2561, and was released by Nokia\cite{10}; TSL2563, on the other hand, was implemented by a single developer. Moreover, the two implementations share no similarity, as CCFinder does not detect any clone between them. This example shows the distributed development nature of Linux, and how drivers released by manufacturers tend to subsume drivers developed by the open source community.

### 4.4 Optional feature to kernel parameter

In this evolution pattern, whose structure is presented in Fig. 10, an optional feature \( F \) is removed from the feature model, but continues to exist in the source code. The key aspect of this pattern relies in its build rules. Originally, the presence of a feature \( F \) defines a new symbol name (macro) that is appended to the macro namespace of the source code under compilation. Such symbol \( (X) \) conditions a block of code \( S \). After the change, \( F \) is removed as a feature and it is turned into a kernel parameter \( F:\text{param} \) that conditions the execution of \( S \) during runtime. In that case, the build rule defining symbol \( (X) \) is dropped.

```
cfex d cpp -dn rt3090 -is -dn rt2860 -w f-w-g+
```

As shown above, the GNU C compiler allows macros to be defined through the -D switch. In our instance, the CONFIG\_PNP\_DEBUG feature was replaced by the boot parameter pnp.debug.

### Discussion

This pattern shows how intricate the Linux kernel three dimensional space is. As illustrated by our instance, the variability switches from being statically compiled to being determined during runtime. Since no functionality is lost and behaviour is preserved, this change results in a software refinement. For the same reasons argued before, evolution occurs in such a way not predicted by existing theory\cite{5}.

### 5. Threats to Validity

The major threat to our work is the incompleteness associated with the analysis of commit logs. Our set of inspected commits resulting in features being removed from the configuration space required us to grep associated commits to have a broader picture of the evolution in place. As this process may fail to recover all associated commits, there is a threat that our evolution patterns reflect a partial view of the real changes. This is why we only present the findings as a preliminary sample of patterns. Further experiments will have to broaden the catalog towards completeness and identify whether these patterns are indeed common.

Furthermore, our analysis is ultimately based on the manual inspection over commits to extract the patterns herein presented. As this process contain certain subjectivity, our patterns may not capture the full intention as envisioned by the original patch authors. To alleviate this, we present concrete instances of each pattern to allow readers to judge whether they reflect the presented structure.

### 6. Related Work

Existing research has already studied the Linux kernel variability. She et al.\cite{14} and Lotufo et al.\cite{8} analyse, among other things, how the Linux variability model evolves...
in terms of feature addition and removal. As we argued in this paper, an analysis based on a single space is incomplete and possibly misleading; features that are no longer present in the variability model do not necessarily cease to evolve, as they might be merged into other features, migrated to implementation space, etc.

Other researchers [5] study the formal aspects of software product line refinement, deriving an evolutionary theory. Such formalism assumes that changes are safe, i.e., do not affect behaviour nor prevents instantiating existing products. Our work shows that Linux does not follow a safe evolutionary model, as certain features are truly removed along the way. Although the authors do not claim completeness, we found real refinement patterns that cannot be explained by their set of theorems.

Borba et al. [4] and Neves et al. [12] provide a catalog of safe transformation templates that, different from ours, do not cover variability evolution when features are removed from the configuration space. In [12], the authors provide evidence on how frequent their templates occur by analyzing the evolution of two small software product lines.


Berger et al. [3] compare Kconfig with other variability modeling languages, such as eCos CDL\(^{16}\) and standard FODA notation. She and Berger [13] study the semantics of Kconfig and its approximation to propositional logic.

Other studies [1, 2] apply static analysis techniques in Makefiles of Linux and FreeBSD to extract feature-to-code mappings.

7. CONCLUSION

We presented a preliminary catalog of evolution patterns extracted from the Linux kernel repository, and explained each pattern in a comprehensive manner, including (but not restricted to) structure, concrete instances and the mechanisms used by developers in achieving them.

Our study is the first to provide explanations on how variability simultaneously evolves in the implementation, compilation and configuration spaces when removing features from the variability model, while keeping them as part of the software. Furthermore, we rely on a complex and variant rich subject of analysis: the Linux kernel.

As future work, we aim to execute a longitudinal study of the Linux kernel to assess the frequency of the patterns we found, along with the discovery of new ones. To allow generalization, we plan to perform similar studies in different software product lines, possibly from different domains.

8. REFERENCES


\(^{16}\)sourceware.org


Challenges in the Evolution of Model-Based Software Product Lines in the Automotive Domain

Hannes Holdschick
Daimler AG
Wilhelm-Runge-Str. 11
89081 Ulm, Germany
hannes.holdschick@daimler.com

ABSTRACT
Using the methodology of software product lines, it is possible to generate program variants with a common core and additional variable modules. Feature-based variant management is especially suitable for documenting differences and similarities of such variants. A variant model created initially quickly becomes obsolete because of the permanent evolution of software functionalities in the automotive area. This is why we need a comprehensive concept how to handle evolution in variant-rich model-based software systems.

In order to achieve this, an exact understanding of the evolution of implementation artifacts is necessary in order to be able to adjust variant modeling for the most important change cases beforehand. This work presents a collection of relevant changes in a functional block model with the necessary adaptation of the variant model.

Categories and Subject Descriptors
D.2.7 [Distribution, Maintenance, and Enhancement]: Restructuring, reverse engineering, and reengineering; D.2.13 [Reusable Software]: Domain Engineering

General Terms
Design

1. INTRODUCTION
The increase in vehicle and functional diversity in the automotive industry leads to an elevated variant complexity of the software systems involved. The concept of software product lines supports the development and mastery of variant-rich, software-based systems [3]. The functional properties of these systems can be expressed using features [6].

In this paper we are concentrating on software variability in model-based function development. In this process, a functional block model will be created which we call a functional model from now on. If the variability reaches a sufficient complexity in a functional model, it should be documented externally in a variant model. A widespread method for this is the feature model [6]. Additionally the configuration knowledge, namely a mapping between features and implementation artifacts, is important for building system variants. In our approach the variant model consists of a feature model and the configuration knowledge.

If a variant model is created, it exists simultaneously next to the functional model, which is subject to constant evolution. Therefore one of the fundamental properties of the variant model is maintainability. Our definition of the term is based on the ISO guideline 9126 on software quality [5]:

Definition 1. The maintainability of a variant model describes its ability to be modified. This includes corrections, improvements, or adaptations of the model to environmental changes.

Considering this definition we can derive quality criteria like easily recognized patterns, understandable structures for rules and dependencies, or small risk for an inconsistent or incorrect model after a change. Which of these properties contributes more or less to the maintainability of the variant model depends on which type of change to the model is common or is understood as complicated. This is why we want to get an overview of which changes relevant to variability could occur in the functional model and where the challenges are when reproducing them in the variant model.

Section 2 describes the important artifacts in this process and section 3 lists three examples of evolutionary change cases in the functional model. The driver assistance system is our example of an evolutionary functionality.

2. FUNDAMENTALS
In this section we describe the structures of feature-oriented software development which are important to us.

2.1 Functional Models
Our considerations on the variability in the implementation domain are based on a model-based functional development approach, as it is known through Matlab / Simulink, Ascet, or similar tools. Accordingly, our functional model can be segmented into a set of components, which are hierarchically structured.

Various mechanisms exist in order to accommodate variability in a component-based model, for example the 150%-approach or delta-oriented programming [8]. Nonetheless, all methods have in common that variation points in the model must be created. This is why we are taking a closer
look at two important possibilities to develop a variation point:

Variation point using an **optional component**: The implemented functionality in an optional component can be activated or deactivated in the model. Either the sub-model with the appropriate functionality is included in the component or a sub-model which lets the component’s input signals pass without further processing.

Example for an optional component: The cruise control (CC) component in Figure 1 constantly maintains a desired speed.

Variation point using **alternative components**: In this case an empty component shell exists, for which a finite number of sub-models are available that implement the alternative functionalities. Exactly one of these sub-models is ultimately included in the component.

Example for alternative components: For cruise control, various operating concepts are offered. This is why the connections to the operating units can be implemented as alternative components, so that the function can be controlled either using one out of two cruise control lever or steering wheel buttons.

A sample functional model is shown in Figure 1, including the optional components Cruise Control (CC), Distance Warning (DW), which gives the driver an optical warning as soon as the distance to the vehicle in front becomes too small, and Brake Assist (BA), which can apply the brakes itself in an emergency.

### 2.2 Feature Models

In the feature model, the functional properties of the domain are structured hierarchically as a tree. Based on common approaches [2, 4], we differentiate the following variation types: mandatory, optional, alternative and or. Additionally, cross-tree constraints also restrict the possible variants, e.g. implication or mutual exclusion. For instance, if we assume, that for security reasons any vehicle with a cruise control also needs the distance warning function, this relation would be expressed using the constraint Crus\(e\)e Control \(\Rightarrow\) Distance Warning (Figure 2).

### 3. CHANGE CASE DESCRIPTION

In the automotive area, the applied software is subject to constant evolution. The following change cases describe three examples and the necessary adaptation as a reaction within the variant model.

#### 3.1 Delete Optional Component

**3.1.1 Description**

CC 1: An optional component exists in the system which represents a variation point. Since the functionality of this component is no longer necessary from now on, we delete it from the functional model which also cancels the variation point.

**3.1.2 Example**

The driver assistance system contains the distance warning component. Meanwhile a new version of the brake assist component has been developed, which takes on the distance warning function. Thus, the distance warning component has become obsolete and is deleted. Before we delete the feature in the feature model, we have to reformulate or delete the existing cross-tree constraint Crus\(e\)e Control \(\Rightarrow\) Distance Warning. Since the distance warning functionality is now part of the brake assist component, we reformulate the constraint as Crus\(e\)e Control \(\Rightarrow\) Brak\(e\)e Assist.

**3.1.3 Modeling**

1. Delete or reformulate the cross-tree constraints affecting the feature, which represents the obsolete component
2. Delete the feature in the feature model

#### 3.2 Optional Component Becomes Mandatory

**3.2.1 Description**

CC 2: There is a optional component located in the system which represent a variation point. The functionality of this component will be integrated into each software system from now on. This component will then become obligatory and the variation point is dropped.

**3.2.2 Example**

The component in Figure 1, which is responsible for the brake assist function used to be optional. We assume, that from now on, every vehicle is to be delivered with brake assist due to security reasons. Thus the component becomes obligatory and is now a part of every driver assistant system (Figure 3). In the feature model we have to deal with the cross-tree constraint Crus\(e\)e Control \(\Rightarrow\) Brak\(e\)e Assist. Since this constraint was not technical we will not keep it in the feature model for documentation.

**3.2.3 Modeling**

1. Delete or reformulate the relations to the corresponding features
2. Delete the corresponding feature in the feature model or set the variation type of the feature to mandatory to better structure the feature model or to guarantee the complete documentation of the domain

---

Figure 1: A Functional Model

Figure 2: the Feature Model belonging to the Functional Model in Figure 1
3.3 New Alternative Component

3.3.1 Description
In this case, a newly implemented functional alternative is added to an existing component. In this process, three subcases can be distinguished depending on the type of the components that already existed.

- **CC 3.1**: If the existing component was obligatory, a new variation point using alternative components will be able to integrate either the existing or the new components into the system.
- **CC 3.2**: If the existing component was optional, a variation point already exists. In this case, there are two possible situations. The existing variation point could be extended with an variation point using alternative components so that a nesting occurs. The new variation point lies in the functional model within the existing variation point. If the optional component is activated, one of the alternative components must be selected. If it remains deactivated, the inner variation point using alternative components is then not a part of the functional model (Situation 1). In the second possible situation, the existing variation point using an optional component is replaced by an variation point using alternative components. The functionality of the optional component remains in place, but it is from then on part of one of the two alternative components on the new variation point (Situation 2).
- **CC 3.3**: If the existing component was already part of a group of alternative components, two distinct situations can arise. Either the new component constitutes a new alternative to all already existing components, or it is solely an alternative for one of the existing components. In the first case, the existing variation point obtains an additional characteristic (Situation 1). In the second case, a new alternative variation point originates in one of the alternative components (Situation 2).

3.3.2 Example
Until now, there was only one possibility to regulate the cruise control, namely the cruise control lever. As a consequence, the regulation was a part of the component that implemented the actual functionality of the cruise control. With a new operating concept using steering wheel buttons, an alternative to the cruise control lever arises, so that the operating concepts for cruise control are outsourced and a variation point, which indicates this new option, comes in addition. The component operating concept (OC) in Figure 4 illustrates this variation point. This development is possible in both cases, no matter whether the existing component, meaning the cruise control, was optional or obligatory. In the feature model we insert a new feature for the operating concept component with two alternative child features representing the two new functional alternatives.

During development, the cruise control lever is, however, improved so that the desired speed can be achieved in small (1 km/h) and large (10 km/h) increments. This two-step cruise control lever is the third control alternative and the variation point is extended accordingly. In the feature model we add two features as children of Cruise_Control_Lever to indicate, that this component has two new alternatives (Figure 6).

3.3.3 Modeling

1. In the case that the existing components were obligatory:
   (a) If no feature exists for the obligatory component yet, one must be inserted
   (b) In this case, there are two possibilities for modeling: Either two new alternative features are added below the existing features for the obligatory features, or the variation type of the existing features is set as alternative and a new alternative feature is inserted in addition.
   (c) Create cross-tree constraints involving other features where appropriate

2. In the case that the existing component was optional:
   (a) Situation 1: Add two new alternative features beneath the existing features for the optional components
   (b) All relations that used to refer to the optional feature must be checked. The constraints that are relevant for both the new alternatives can continue to refer to the optional feature, if it still exists. If a constraint is only relevant for one of the new alternatives, then the optional feature must be replaced by the corresponding alternative feature when adapting the constraint.
   (c) Create cross-tree constraints involving other features where appropriate

3. In the case that the existing component was alternative:
Figure 5: Final Version of our Sample Functional Model

(a) Situation 1: A new alternative feature is added in addition to the already existing alternative features.
Situation 2: Two new alternative features are added as the child of an already existing alternative feature.
(b) Create cross-tree constraints involving other features where appropriate

4. RELATED WORK

The refactoring catalog described by Alves et al. [1] is partially relevant to our work, since they describe feature model refactorings in the context of software product lines as changes that do not restrict the configurability of the model. In our consideration of evolution, arbitrary changes are allowed with respect to the configurability of the model.

In another approach [9], changes in the feature model are classified into the four classes refactoring, specialization, generalization, or arbitrary change. As mentioned above, all four classes are relevant to us; however, our work is more directed towards a description of exact incremental variant model adaptations in order to reproduce a changed situation in implementation.

The analysis of the Linux kernel by Lotufo et al. shows that the evolutionary steps described here can also appear in other systems [7]. In this work preserving the consistency of implementation and variation models is described as a major challenge as well, which confirms what we outlined in our problem description.

5. CONCLUSION AND FUTURE WORK

The consistency between the variant model and the actual variability situation in its implementation is a major challenge in the industrial sector. A variant model created initially quickly becomes a part of the functional model’s evolution after its integration into the development process and for this reason must exhibit maintainable structures that ease adaptations. For a better understanding of the evolutionary steps in a model-based implementation artifact, this work describes such important steps, the corresponding adaptations of the variant model, and the challenges that appear in the process.

In our future work, we will expand the collection of further developments in the functional model. The experience we collected will be the basis for modeling guidelines and consistency conditions for a maintainable variant model. In order to ease the work of the developers modifying variant models, we strive towards automatic implementation of more complex evolutionary steps.

Figure 6: the Feature Model belonging to the Functional Model in Figure 5

Acknowledgement

The content of this work is partially funded by the German ministry of education and research (BMBF) in the context of the project SPES XT (No. 01IS12005P).

6. REFERENCES