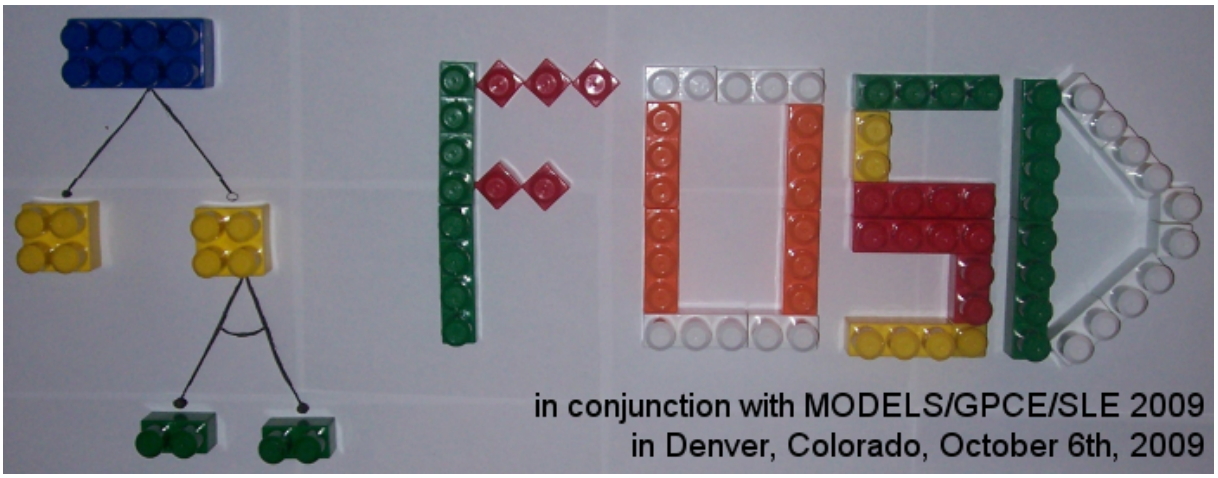


Proceedings of the First International Workshop on Feature-Oriented Software Development (FOSD)

October 6, 2009
Denver, Colorado, USA

Editors: Sven Apel *(University of Passau, DE)*
William R. Cook *(University of Texas at Austin, US)*
Krzysztof Czarnecki *(University of Waterloo, CA)*
Christian Kästner *(University of Magdeburg, DE)*
Neil Loughran *(SINTEF, NO)*
Oscar Nierstrasz *(University of Berne, CH)*

Proceedings published online in the ACM Digital Library

www.acm.org

Printed proceedings sponsored by Metop GmbH

www.metop.de

The Association for Computing Machinery
2 Penn Plaza, Suite 701
New York, New York 10121-0701
U.S.A.

ACM ISBN: 978-1-60558-567-3

Foreword

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, so-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. For example, features of a database system could be transaction management, query optimization, and multi-user operation, those of a banking software could be account management, authentication, and financial transactions, and those of a text processing system could be printing, spell checking, and document format conversion. 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/documents of a modular software system. For example, the feature transaction management would affect many parts of a database system, e.g., query processing, logical and physical optimization, and buffer and storage management. 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 main goal of the FOSD'09 workshop is to foster and strengthen the collaboration between the different researchers who work in the field of FOSD or in the related fields of software product lines, aspect-oriented software development, service-oriented architecture, and model-driven engineering. A keynote by Don Batory, a leading researcher in FOSD, will be an excellent start up for discussions on historical perspectives, current issues, and visions of FOSD.

The FOSD workshop builds on the success of a series of workshops on product lines, generative programming, and aspect orientation, held at GPCE'06, GPCE'07, and GPCE'08. In the predecessor workshops it became apparent that the concept of features and the paradigm of FOSD is central to the thinking of a whole community and is related to the concepts found in different other communities. So, the idea grew to dedicate a workshop specifically to FOSD in order to set a proper focus. Furthermore, four of the organizers of this workshop (Sven Apel, William R. Cook, Krzysztof Czarnecki, and Oscar Nierstrasz) are organizing a research seminar on FOSD at the renowned Dagstuhl castle. The seminar proposal has recently been accepted by Dagstuhl castle and the seminar will take place in January 2011. So, a further motivation for the FOSD workshop is the idea to hold the workshop at MODELS / GPCE / SLE 2009 as a kick-off meeting for the FOSD Dagstuhl Seminar.

Sven Apel
William R. Cook
Krzysztof Czarnecki
Christian Kästner
Neil Loughran
Oscar Nierstrasz

Program Committee

Vander Alves (University of Brasilia, BR)
David Benavides Cuevas (University of Seville, ES)
Danilo Beuche (pure-systems, DE)
Iris Groher (University of Linz, AT)
Kyo-Chul Kang (POSTECH, KR)
Thomas Leich (Metop Research Institute, DE)
Christian Lengauer (University of Passau, DE)
Roberto Lopez-Herrejon (Bournemouth University, UK)
Klaus Ostermann (University of Aarhus, DK)
Susanne Patig (University of Berne, CH)
Christian Prehofer (Nokia Research, FI)
Olaf Spinczyk (University of Dortmund, DE)
Christine Schwanninger (Siemens, DE)
Salvador Trujillo (IKERLAN Research Centre, ES)

Table of Contents

Keynote

On the Importance and Challenges of FOSD 1
Don Batory

Session 1: Languages & Product Derivation

Language Support for Feature-Oriented Product Line Engineering 3
Wonseok Chae and Matthias Blume

Feature-Oriented Programming with Ruby 11
Sebastian Günther and Sagar Sunkle

Remodularizing Java Programs for Comprehension of Features 19
Andrzej Olszak and Bo Nørregaard Jørgensen

An Orthogonal Access Modifier Model for Feature-Oriented Programming 27
Sven Apel, Jörg Liebig, Christian Kästner, Martin Kuhlemann, and Thomas Leich

Product Derivation for Solution-Driven Product Line Engineering 35
Christoph Elsner, Daniel Lohmann, and Wolfgang Schröder-Preikschat

A Model-Based Representation of Configuration Knowledge 43
Jorge Barreiros and Ana Moreira

Session 2: Experience Reports & Correctness

Domain analysis on an Electronic Health Records System 49
Xiaocheng Ge, Richard Paige, and John McDermid

How to Compare Program Comprehension in FOSD Empirically – An Experience Report 55
Janet Feigenspan, Christian Kästner, Sven Apel, and Thomas Leich

RobbyDBMS – A Case Study on Hardware/Software Product Line Engineering 63
Jörg Liebig, Sven Apel, Christian Lengauer, and Thomas Leich

Towards Systematic Ensuring Well-Formedness of Software Product Lines 69
Florian Heidenreich

An Extension for Feature Algebra 75
Peter Höfner and Bernhard Möller

Dead or Alive: Finding Zombie Features in the Linux Kernel 81
Reinhard Tartler, Julio Sincero, Wolfgang Schröder-Preikschat, and Daniel Lohmann

Session 3: Model-Driven Development

Feature-Oriented Refinement of Models, Metamodels and Model Transformations 87
Salvador Trujillo, Ander Zubizarreta, Xabier Mendialdua, and Josune De Sosa

Model-Driven Development of Families of Service-Oriented Architectures 95
Mohsen Asadi, Bardia Mohabbati, Nima Kaviani, Dragan Gasevic, Marko Boskovic, and Marek Hatala

Interaction-based Feature-Driven Model-Transformations for Generating E-Forms 103
Bedir Tekinerdogan and Namik Aktekin

Towards Feature-driven Planning of Product-Line Evolution 109
Götz Botterweck, Andreas Pleuss, Andreas Polzer, and Stefan Kowalewski

Detecting Feature Interactions in SPL Requirements Analysis Models 117
Mauricio Alferez, Ana Moreira, Uirá Kulesza, Joao Araujo, Ricardo Mateus, and Vasco Amaral

Appendix

Author Index 125

On the Importance and Challenges of FOSD

Don Batory
Department of Computer Science
University of Texas at Austin, USA
batory@cs.utexas.edu

ABSTRACT

Among the key elements of mature engineering is automated production: we understand the technical problems and we understand their solutions; our goal is to automate production as much as possible to increase product quality, reduce costs and time-to-market, and be adept at creating new products quickly and cheaply.

Automated production is a technological statement of maturity: "We've built these products so often by hand that we've gotten it down to a Science". Models of automated production are indeed the beginnings of a *Science of Automated Design (SOAD)*. *Feature Oriented Software Development (FOSD)* will play a fundamental role in SOAD, and I believe also play a fundamental role in the future of software engineering.

In this presentation, I explain what distinguishes FOSD from other software design disciplines and enumerate key technical barriers that lie ahead for FOSD and SOAD.

Categories and Subject Descriptors: D.2.2 [**Software**]: Software Engineering—*Design Tools and Techniques*; D.2.11 [**Software**]: Software Architectures—*Domain-specific Architectures*; D.3.3 [**Software**]: Programming Languages—*Language Constructs and Features*.

General Terms: Design, Languages, Theory.

Keywords: Features, Verification, Testing, Feature Interactions, Feature-Oriented Software Development, Science of Automated Design.

FOSD'09, October 6, 2009, Denver, Colorado, USA.

Language Support for Feature-Oriented Product Line Engineering

Wonseok Chae
Toyota Technological Institute at Chicago
wchae@tti-c.org

Matthias Blume
Google, Inc.
blume@google.com

ABSTRACT

Product line engineering is an emerging paradigm of developing a family of products. While product line analysis and design mainly focus on reasoning about commonality and variability of family members, product line implementation gives its attention to mechanisms of managing variability. In many cases, however, product line methods do not impose any specific synthesis mechanisms on product line implementation, so implementation details are left to developers. In our previous work, we adopted feature-oriented product line engineering to build a family of compilers and managed variations using the Standard ML module system. We demonstrated the applicability of this module system to product line implementation. Although we have benefited from the product line engineering paradigm, it mostly served us as a design paradigm to change the way we think about a set of closely related compilers, not to change the way we build them. The problem was that Standard ML did not fully realize this paradigm at the code level, which caused some difficulties when we were developing a set of compilers.

In this paper, we address such issues with a language-based solution. **MLPolyR** is our choice of an implementation language. It supports three different programming styles. First, its first-class cases facilitate composable extensions at the expression levels. Second, its module language provides extensible and parameterized modules, which make large-scale extensible programming possible. Third, its macro system simplifies specification and composition of feature related code. We will show how the combination of these language features work together to facilitate the product line engineering paradigm.

Categories and Subject Descriptors

D.2.13 [**Software Engineering**]: Reusable Software—*domain engineering*; D.3.2 [**Programming Languages**]: Language Classifications—*applicative (functional) languages, extensible languages, multiparadigm languages*

FOSD'09, October 6, 2009, Denver, Colorado, USA.

General Terms

Design, Languages

Keywords

Feature-Oriented Programming, Product line engineering

1. INTRODUCTION

Product line engineering is a paradigm of developing a family of products [19, 22]. This emerging paradigm encourages developers to focus on developing a set of products rather than on developing one particular product. Therefore, we are expected to develop products from a common set of components (*core assets*) rather than from scratch. So far most efforts of working under this paradigm have focused on how to analyze a family of products and develop reusable assets. Features are commonly used to reason about commonality and variability in product lines. A set of features define a design space and the selection of a particular subset is the first step towards the synthesis of a design artifact [18].

In our previous work, we demonstrated that the product line engineering as a developing paradigm was a very effective way to build a family of compilers [12]. We showed engineering activities from product line analysis through product line architecture design to product line component design. Then, we presented how to build particular compilers from core assets resulting from such domain engineering activities. This approach especially helped us maintain source code by providing a systematic way to identify variations. This variability analysis provided us with the better chance of utilizing underline implementation technology. For example, the product line analysis enabled us to predict which parts would be mostly changeable, so we could parameterize over them via functorization provided by the Standard ML [5]. Then, the main program became a functor which took variations and produced a specific compiler. That is, we obtained a compiler generator. Product line engineering as a design paradigm was truly helpful here in that its product line analysis acts as a systematic way of identifying commonalities and variabilities. This made it possible to design and implement reusable and flexible software by supplying guidance on where relevant implementation techniques could be applied.

Although we have benefited from the product line engineering paradigm, it mostly served us as a design paradigm to change the way we think about a set of closely related compilers, not to change the way we build them. The problem was that our implementation technology did not fully

realize this paradigm at the code level. As a result, we experienced some difficulties:

- The relations among features and core assets (i.e., architecture and component models) were implicitly expressed only during the product line analysis as a form of documentation. Other product line model-based methods usually provide a way to express those relations explicitly by using CASE tools. In FORM, for example, those explicit relations make it possible to automatically generate product code from specifications [18].
- Our approach poorly supported systematic generation of family members.
- While we demonstrated our underline implementation technology (i.e., the Standard ML module system) was powerful enough to manage variations in the context of product lines, its type system sometimes imposed restrictions which caused code duplication between functions on data types.

This paper proposes a language support approach to address such limitations. Firstly, we will give a brief overview of the feature-oriented product line engineering (Section 2). As in our previous work, we will start with the feature modeling followed by product line asset design activities. Then, we will review the above issues in the context of product line implementation. Then, we present a language-based solution to better support product line implementation (Section 3). Our choice of an implementation language is **MLPolyR** [10]. It supports extensible programming in a compositional way and its module language provides parameterization mechanisms. Furthermore, its macro system provides feature-related code composition which is based on the feature selection. We will show how the combination of these language features work together to facilitate the product line engineering paradigm. Finally, we will review other feature-oriented implementation approaches (Section 4).

2. PRODUCT LINE ENGINEERING

Product line engineering highlights the development of products from core assets rather than from scratch. Therefore, this paradigm separates the process of building core assets from the process of building an individual product as shown in Figure 1. Product line asset development consists of analyzing a product line, designing reference architectures, and developing reusable components. In the product development process, a particular product is instantiated by selecting a proper architecture and adapting components. Among various product line approaches, we adopt FORM product line engineering for the following reasons:

- The method relies on a feature-based model which provides adequate means for reasoning about product lines [19].
- The method supports architecture design which plays an important role in bridging the gap between the concepts at the requirement level and their realization at the code level by deciding how variations are modularized by means of architectural components [25].

In this section, we will give an overview of overall engineering activities for developing a family of the simple arithmetic

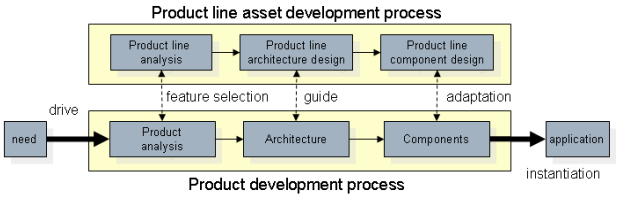


Figure 1: Development process (adopted from FORM [19]).

language interpreters. We believe that this simplified example can demonstrate the same limitation that our previous experience showed without dealing with unrelated technical details. Furthermore, this example is designed to precisely capture so-called *the expression problem* [29], which can even facilitate a comparative study of language support for product line implementation. The more detailed discussion of our choice of this example and the working draft of this comparative study can be found at our technical report [13].

2.1 Problem description

Let us consider a Simple Arithmetic Language (SAL) that contains terms such as numbers, variables, additions and a let-binding form. Suppose we start with two basic operations on terms: a function eval that realizes the evaluation semantics and a function check that realizes the static semantics. (In this case the static semantics just makes sure that all variables are in scope.)

In this setup, assume one wants to build an SAL interpreter I, which is the composition of the combinators eval and check where o means function composition:

$$\mathsf{I} = \mathsf{eval} \ \circ \ \mathsf{check}$$

Sometimes, we want to replace an old implementation with a new one. For example, instead of the evaluation semantics eval, we can define a machine semantics (for example because we want to make control explicit) and implement its realization (eval_m):

$$\mathsf{I}_\mathsf{m} = \mathsf{eval}_\mathsf{m} \ \circ \ \mathsf{check}$$

Optionally, the combinator opt which performs some simple term rewriting (e.g., constant-folding, strength-reduction, etc.) may be inserted to build an optimized interpreter I_opt:

$$\mathsf{I}_\mathsf{opt} = \mathsf{eval}_\mathsf{m} \ \circ \ \mathsf{opt} \ \circ \ \mathsf{check}$$

When the base language grows to support additional terms (e.g., a conditional term), eval, opt and check also evolve to constitute a new interpreter I'_opt:

$$\mathsf{I}'_\mathsf{opt} = \mathsf{eval}' \ \circ \ \mathsf{opt}' \ \circ \ \mathsf{check}'$$

Since all these interpreters have so much in common, we should be able to understand them as a *family* of interpreters. Therefore, it is natural to apply product line engineering for better support of their development.

2.2 Feature-oriented product line engineering

To analyze a set of interpreters as a family, we adopt feature-oriented product line engineering, which consists of following engineering activities:

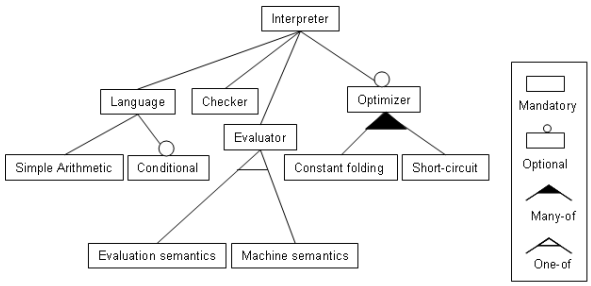


Figure 2: Feature model for the SAL interpreter. A closed triangle (many-of) represents multiple optional relationship and an open triangle (one-of) represents alternative relationship [15].

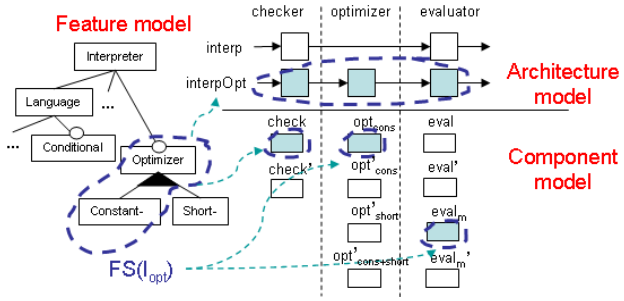


Figure 3: The overall product engineering process. Each selected feature gives advice on how to choose an architecture and how to instantiate required components. For example, the reference architecture interpOpt gets selected, guided by the presence of the Optimizer feature. The presence of the Constant folding feature guides us to choose the component optimizer with the implementation $\mathsf{opt}_{\mathsf{cons}}$ [13].

- *Product line analysis.* We perform commonality and variability analysis for the family of the SAL interpreters. We can easily consider features in the base interpreter as commonalities and exclusive features only in some extensions as variations. Based on this analysis, we identify two kinds of variations: architectural variation and component level variations. Then we determine which factors cause these variations. Such factors are represented as *features* in the feature model as illustrated in Figure 2.

- *Product line architecture design.* Architecture design involves identifying conceptual components and specifying their configuration. During this phase, we have to not only identify components but also define interfaces between components:

$$\begin{array}{lcccl} \mathsf{checker} & : & \mathsf{term} & \rightarrow & \mathsf{term} \\ \mathsf{optimizer} & : & \mathsf{term} & \rightarrow & \mathsf{term} \\ \mathsf{evaluator} & : & \mathsf{term} & \rightarrow & \mathsf{value} \end{array}$$

  As usual, the arrow symbol $\rightarrow$ is used to specify a function type. In our example, components act like pipes in a pipe-and-filter the architectural style, so all interface information is captured by the type.

  By using the above conceptual components, we can specify the overall structure (i.e., *architecture*) of various interpreters:

$$\begin{array}{lcl} \mathsf{interp} & = & \mathsf{evaluator} \circ \mathsf{checker} \\ \mathsf{interpOpt} & = & \mathsf{evaluator} \circ \mathsf{optimizer} \circ \mathsf{checker} \end{array}$$

- *Product line component design.* Next, we identify conceptual components which are constituents of a conceptual architecture. A conceptual component can have multiple implementations. For example, there are many versions of the evaluator component depending on the evaluation strategy:

$$\begin{array}{lcccl} \mathsf{eval} & : & \mathsf{term} & \rightarrow & \mathsf{value} \\ \mathsf{eval}_{\mathsf{m}} & : & \mathsf{term} & \rightarrow & \mathsf{value} \end{array}$$

  At the same time, the language term can be extended to become term' which is an extension of term (for example to support conditionals):

$$\begin{array}{lcccl} \mathsf{eval}' & : & \mathsf{term}' & \rightarrow & \mathsf{value} \\ \mathsf{eval}'_{\mathsf{m}} & : & \mathsf{term}' & \rightarrow & \mathsf{value} \end{array}$$

  Similarly, check and check' can be specified as follows:

$$\begin{array}{lcccl} \mathsf{check} & : & \mathsf{term} & \rightarrow & \mathsf{term} \\ \mathsf{check}' & : & \mathsf{term}' & \rightarrow & \mathsf{term}' \end{array}$$

  For the optimizer component, there are many possible variations due to inclusion or exclusion of various individual optimization steps (i.e., constant folding and short-circuiting) and due to the variations in the underlying term language (i.e., base and extended): $\mathsf{opt}_{\mathsf{cont}}$, $\mathsf{opt}'_{\mathsf{cons}}$, $\mathsf{opt}'_{\mathsf{short}}$ or $\mathsf{opt}'_{\mathsf{cons+short}}$.

- *Product engineering.* Product engineering starts with analyzing the requirements provided by the user and finds a corresponding set of required features from the feature model. Assuming we are to build four kinds of interpreters, we have to have four different feature selections:

$$\begin{array}{lcl} \mathsf{FS}(\mathsf{I}) & = & \{\mathsf{Evaluation\ semantics}\} \\ \mathsf{FS}(\mathsf{I}_{\mathsf{m}}) & = & \{\mathsf{Machine\ semantics}\} \\ \mathsf{FS}(\mathsf{I}_{\mathsf{opt}}) & = & \{\mathsf{Machine\ semantics}, \mathsf{Optimizer}, \\ & & \mathsf{Constant\ folding}\} \\ \mathsf{FS}(\mathsf{I}'_{\mathsf{opt}}) & = & \{\mathsf{Conditional}, \mathsf{Evaluation\ semantics}, \mathsf{Optimizer}, \\ & & \mathsf{Constant\ folding}, \mathsf{Short-circuit}\} \end{array}$$

  Here, the function FS maps a feature product to its corresponding set of its required features. (For brevity only non-mandatory features are shown.) Selected feature sets give advice on the selection among both reference architectures and components. Figure 3 shows the overall product engineering process. The target product would be instantiated by assembling such selections.

2.3 Issues in product line implementation

During the product line asset development process, we obtain reference models which represent architectural and component level variations. Such variations should be realized at the code level. The first step is to refine conceptual architectures into concrete architectures which describe how to configure conceptual components. Then, product

line component implementation involves realization of conceptual components with the proper product feature delivery methods. Based on our experience of building a family of compilers [12], we see three issues that surfaced during product line implementation.

- *Product line architecture implementation.* Since there may be multiple reference architectures, it would be convenient to have mechanisms for abstracting architectural variations, capturing the inclusion or exclusion of certain components. Hence, any adequate implementation technique should be able to provide mechanisms for:
  - declaration of required conceptual components (checker, optimizer and evaluator) and their interface
  - specification of the base reference architecture interp and its optimized counterparts interpOpt by using such conceptual components.

  Our experience showed that the Standard ML (SML) module language was powerful enough to describe both components and their interfaces. Each components in a reference architecture was mapped into a SML program unit (called *structure*) and interfaces among them as *signatures*. Furthermore, *functors* make it possible to implement the common part once and parameterize variations so that different products can be instantiated by assigning distinct values as parameters:

```
functor InterpFun(structure C : CHECKER
                  structure E : EVALUATOR
                  sharing C.T = E.T) : INTERP
where type term = E.T.term =
struct
      type term = E.T.term
      val interp = E.eval o C.check
end
```

  Here, the functor InterpFun takes two components checker and evaluator and then returns their composition. However, all necessary sharing constraints between parameters must be declared explicitly. For example, in order to avoid type errors we had to specify that the types of term in both components should coincide.

- *Product line component implementation.* This phase involves realization of conceptual components. The main challenge of this phase is how to implement variations at the component level. Such variations could be in the form of either code extension or code substitution. For our running example, a pair of check and check′ corresponds to code extension while a pair of eval and eval_m corresponds to code substitution. While its parameterized module (i.e., functor) provides an efficient way to implement code substitution, the SML type system sometimes imposes restriction on code extension, which causes code duplication between functions on data types.

- *Product engineering.* Based on the product analysis, a feature product is instantiated by assembling product line core assets. For example, an extended interpreter I′ can be instantiated by applying the functor InterpFun to the modules EChecker (realizing check′) and EBigStep (realizing eval′) based on the feature selection. Each selected feature gives advice on how to choose an application architecture and how to instantiate required components. However, such instantiation was manually performed since the relations between features and core assets are implicitly expressed as a form of documentation.

3. LANGUAGE SUPPORT FOR PRODUCT LINE IMPLEMENTATION

In this section, we present product line implementation using **MLPolyR** as a language-based solution. **MLPolyR** is an ML-like language with row polymorphism, polymorphic record selection and polymorphic sums, functional record update and a Hindley-Milner-style type system with principal types [10, 11]. Among its many features, we focus on three aspects in this paper:

- *The extensible module language.* Similar to functors in the Standard ML module system, MLPolyR provides an parameterized mechanism called *template*. Furthermore, its modules are extensible and compiled separately.
- *Extensible programming with first-class cases.* With cases being first-class and extensible, one can use the usual mechanisms of functional abstraction in a style of programming that facilitates composable extensions. Note that these composable extensions are type-safe in a sense that well-typed programs do not go wrong [24].
- *The macro system.* Recent addition to **MLPolyR** supports code expansion at the level of a macro system. This mechanism makes it possible to write composition specification in terms of features, then feature selection will integrate the corresponding code easily.

In the remainder of this section, we will show how such mechanisms resolve each issue previously identified.

3.1 Product line architecture implementation

Each component in a reference architecture is mapped to an **MLPolyR** module. We first define types (or signatures) of the interested components based on the outcome of product line architecture design:

```
Checker   : {{ check : term → term,  ... }}
Optimizer : {{ opt   : term → term,  ... }}
Evaluator : {{ eval  : term → int,   ... }}
```

where ... implies that there may be more parts in a component, but they are not our concerns. In practice, we do not have to write such interface explicitly since the type checker infers the principal types. Then, by using these conceptual modules (Checker, Optimizer and Evaluator), we can define two reference architectures:

```
module Interp = {{
    val interp = fn e => Evaluator.eval
                           (Check.check e)
}}

module InterpOpt = {{
    val interp = fn e => Evaluator.eval
                           (Optimizer.opt
                             (Checker.check e))
}}
```

Alternatively, like functors in SML, we can use a parameterized module called a *template* which takes concrete modules as arguments and instantiates a composite module:

```
1  template InterpFun (C, E) = {{
2      val interp = fn e => E.eval (C.check e)
3  }}
4
5  template InterpOptFun (C, O, E) = {{
6      val interp = fn e => E.eval
7                             (O.opt
8                               (C.check e))
9  }}
```

where C, O and E implicitly imply Checker, Optimizer and Evaluator respectively and their signatures are captured as constraints by the type checker. For example, the type checker infers the constraint that the module C should have a component named check which has a type of $\alpha \to \beta$ and β should be either an argument type of the module E (Line 1) or that of O (Line 5).

The second approach with templates supports more code reuse because a reference architecture becomes polymorphic because it is parameterized over its components including their types. As long as components satisfy constraints that the type checker computes, any components can be plugged into a reference architecture. For example, for the argument C, either the base module Check and its extension EChecker can applied to the template InterpFun.

3.2 Product line component implementation

Modules in **MLPolyR** realize components. In order to manage component-level variations, we have to deal with both code extension and code substitution. For example, we will see multiple implementations of the component Evaluator:

$$\begin{array}{lcl} \mathsf{BigStep} & : & \{\{\ \mathsf{eval} : \mathsf{term} \to \mathsf{int}, \quad \ldots\ \}\} \\ \mathsf{Machine} & : & \{\{\ \mathsf{eval} : \mathsf{term} \to \mathsf{int}, \quad \ldots\ \}\} \\ \mathsf{EBigStep} & : & \{\{\ \mathsf{eval} : \mathsf{term}' \to \mathsf{int}, \quad \ldots\ \}\} \\ \mathsf{EMachine} & : & \{\{\ \mathsf{eval} : \mathsf{term}' \to \mathsf{int}, \quad \ldots\ \}\} \end{array}$$

where term represents a type of the base constructors and term′ that of the extension. BigStep and EBigStep implement the evaluation semantics and its extension while Machine and EMachine implement the machine semantics and its extension. Note that a pair of BigStep and EBigStep (and also a pair of Machine and EMachine) corresponds to code extension while a pair of BigStep and Machine corresponds to code substitution.

Code extension is supported by first-class extensible cases. Figure 4 shows how such extensions are made. First, we define cases (Line 4-10) separately from a `match` expression (Line 14) where cases will be consumed. Then, we wrap such cases in functions by abstracting over their free variables (i.e., eval and env) (Line 3). One of these variables is eval—the whole evaluator itself. Its inclusion in the argument list achieves open recursion, which is essential to extensibility. With this setup, it becomes easy to add a new case (i.e., IF0). In an extension, only a new case is handled (Line 22-24) and the default explicitly refers to the original set of other cases represented by BigStep.bases (Line 25). Then, EBigStep.bases can handle five cases including IF0. We can obtain a new evaluator EBigStep.eval by closing the recursion through applying bases to evaluator itself (Line 29). Note that a helper function run is actually applied instead of eval in order to pass an initial environment in Line 30.

```
1  (* module for the evaluation semantics *)
2  module BigStep = {{
3      fun bases (eval, env) =
4          cases `VAR x => env x
5                | `NUM n => n
6                | `PLUS (e1, e2) =>
7                    eval (env, e1) + eval (env, e2)
8                | `LET (x, e1, e2) =>
9                    eval(Env.bind
10                       (eval (env, e1), x, env), e2)
11
12     fun eval e =
13         let fun run (env, e) =
14                 match e with bases (run, env)
15         in  run (Env.empty, e)
16         end
17 }}
18
19 (* module for the extended evaluation semantics *)
20 module EBigStep = {{
21     fun bases (eval, env) =
22         cases `IF0 (e1, e2, e3) =>
23             if eval (env, e1) == 0 then
24                 eval (env, e2) else eval (env, e3)
25         default: BigStep.bases (eval, env)
26
27     fun eval e =
28         let fun run (env, e) =
29                 match e with bases (run, env)
30         in  run (Env.empty, e)
31         end
32 }}
```

Figure 4: The module BigStep realizes the evaluation semantics (eval) and the module EBigStep realizes the extended evaluation semantics (eval′) by defining only a new "conditional" case `IF0.

Code substitution as another form of variation at the component level does not cause any trouble. For example, the module Machine implements the machine semantics (i.e., eval_m). In our example two different implementations (BigStep and Machine) provide interchangeable functionality, but neither is an extension of the other, so they are implemented independently.

3.3 Product engineering

Our example asks us instantiate four interpreters (I, I_m, I_opt and I'_opt) differentiated by the feature selection. As Figure 3 demonstrates, each will be instantiated by selecting a proper architecture (either InterpFun and InterOptFun) and choosing its components (either BigStep or Machine, etc) with implicit advice from the selected feature set. For example:

- When the feature set is $\mathsf{FS}(\mathsf{I}_\mathsf{m})$, the reference architecture InterpFun gets chosen. Then, two components Machine and Checker are selected because of the presence of Machine semantics feature. Therefore, we instantiate the interpreter I_m as follows:

  $$\mathsf{module}\ \mathsf{I}_\mathsf{m} \ = \ \mathsf{InterpFun} \quad (\mathsf{Checker}, \mathsf{Machine})$$

- When the feature set is $\mathsf{FS}(\mathsf{I}'_\mathsf{opt})$, the reference architecture InterpOptFun is chosen. As far as the components are concerned, the presence of the Conditional and Evaluation semantics features guide us to choose the component EBigStep. Similarly, the presence of

the Optimizer, Conditional, Constant folding and Short-circuit forces the use of component ECSOptimizer (realizing $\mathsf{opt}'_{\mathsf{cons+short}}$). Therefore, we instantiate the interpreter $\mathsf{I}'_{\mathsf{opt}}$ as follows:

$$\begin{array}{lcll} \mathsf{module}\ \mathsf{I}'_{\mathsf{opt}} & = & \mathsf{InterpOptFun} & (\mathsf{EChecker}, \\ & & & \mathsf{ECSOptimizer}, \\ & & & \mathsf{EBigStep}) \end{array}$$

In this approach, we have to rely on implicit guidance on how to assemble core assets and then we manually perform a product instantiation since conventional languages cannot state the relations between a feature and its corresponding code segments in the program text.However, the **MLPolyR** macro system provides a way to explicitly specify the relations between features and core assets as a set of rules, so we can specify feature configuration in a separate file. One benefit of it is that the **MLPolyR** compiler can automate product engineering process, that is, it can automatically pick a right reference architecture and its associated components with respect to the given rules.

Figure 5 shows the expansion rules for a family of the SAL interpreters. It is used to map feature sets into feature-related abstractions (i.e., module names). For example, suppose the following macro term in the program text:

```
1 module Ix = @Interp
```

The macro term @Interp gets expanded recursively depending on the feature selection at the parse time. Let us assume that the selected feature set equals to $\mathsf{FS}(\mathsf{I}'_{\mathsf{opt}})$. A reference architecture InterpOptFun will get selected (Line 4 in Figure 5) because of the presence of the Optimizer feature. Then, its conceptual component arguments Checker, Optimizer, Evaluator will be recursively instantiated into the proper concrete components. Checker will be expanded into a component ECheck due to the presence of Conditional (Line 10). Optimizer will be expanded into ECSOptimizer due to Optimizer, Conditional, Constant folding and Short-circuit (Line 15). Evaluator will be expanded into EBigStep due to the Conditional and Evaluation semantics features (Line 18). As a result, the term @Interp expands to the following expressions, which equals to the declaration of the module $\mathsf{I}'_{\mathsf{opt}}$ that we had to manually write down:

```
1 InterpOptFun (EChecker, ECSOptimizer, EBigStep)
```

In summary, the **MLPolyR** compiler can expand macro terms with respect to rules, so the feature composition can be done automatically once we write expansion rules and provide a valid feature set.

4. DISCUSSION AND RELATED WORK

Our approach adopts FORM which aims to provide an unified feature-oriented development methodology [18, 19]. This method describes a mapping between features and core assets (i.e., design and implementation artifacts). However, it does not impose any specific synthesis mechanisms on its implementation, so implementation details are left to developers. Its early papers only illustrated the usage of the object-oriented programming annotated by the FORM macro language [18, 17] but recently, aspect-oriented programming has become popular as a way of implementing features in a compositional way [21, 14]. Our prior work on building a family of compilers introduced another choice of implementation mechanism, i.e., functorization [12]. The contribution of this paper lies that we identify some difficulties of our previous approach and propose a direct language support for each phases in the product line implementation process:

- *Product line architecture implementation.* Similar to functors in SML, **MLPolyR** provides a parameterized module which has been demonstrated to be powerful to manage variations. Unlike SML, however, we do not have to write signatures explicitly since the **MLPolyR** type checker infers the principal types, which is expected to lessen the burden of product line implementors.

- *Product line component implementation.* The main challenge of this phase is to provide extensibility mechanism. Here, **MLPolyR** supports first-class cases that facilitate composable extensions.

- *Product engineering.* The **MLPolyR** macro system simplifies feature selection-based composition, so product instantiation can be automated once we supply a valid feature set.

While we aim to map between features in feature modeling and language constructs (realizing features), there have been attempts to support the concept of features more explicitly at code level. AHEAD, FeatureC++, CaesarJ and FeatureHouse are such feature-oriented programming languages that provide better abstraction and modularization mechanisms for features in various ways [8, 4, 6, 2]. Most these existing mechanisms fall into one of three categories:

- *The annotative approach.* This approach implements features using some form of annotations. Typically, preprocessors, e.g., macro systems, have been used in many literature examples. For example, the macro language in FORM determines inclusion or exclusion of some code segments based on the feature selection [18, 17].

- *The compositional approach.* In this approach, features are implemented as distinct units which are then integrated to become a product. Aspect-oriented or mixin-layered extension are such examples [8, 7].

- *The parameterization approach.* The idea of parameterized programming is to implement the common part once and parameterize variations so that different products can be instantiated by assigning distinct values as parameters. Higher-order modules, also known as *functors*—e.g., in SML are a typical example [5]. The SML module system has been demonstrated to be powerful enough to manage variations in the context of product lines [12].

Note that there have been hybrid attempts combining multiple different approaches [3, 20] and the **MLPolyR** language supports all three different programming styles. Similarly, FeatureC++ also supports all of them by combining aspect-oriented programming style, generic programming style (which enables parameterization over refinements) and an annotative approach [4]. However, its annotations are line-based in the style of #ifdef directives. Therefore,

```
1  (* Architecture Model *)
2
3  Interp ::= InterpFun     (Checker, Evaluator)             {}
4           | InterpOptFun (Checker, Optimizer, Evaluator) {Optimizer}
5
6
7  (* Component Model *)
8
9  Checker   ::= Check          ()   {}
10             | ECheck         ()   {Conditional}
11
12 Optimizer ::= COptimizer     ()   {Optimizer, Constant folding}
13             | ECOptimizer    ()   {Optimizer, Conditional, Constant folding}
14             | ESOptimizer    ()   {Optimizer, Conditional, Short-Circuit}
15             | ECSOptimizer   ()   {Optimizer, Conditional, Constant folding, Short-Circuit}
16
17 Evaluator ::= BigStep         ()   {Evaluation semantics}
18             | EBigStep       ()   {Evaluation semantics, Conditional}
19             | Machine        ()   {Machine semantics}
20             | EMachine       ()   {Machine semantics, Conditional}
```

Figure 5: Expansion rules for the SAL product line. ::= denotes a "is-one-of" relation. For example, a macro term @Interp can be implemented by either a template InterpFun or InterpOptFun (Line 3-4). A template can have module arguments in the following (...). A module argument can recursively have its own expansion rules. Each rule defines the corresponding features in the list bracketed by {...}.

their feature specific code segments (if annotated) are scattered across multiple classes, so code easily becomes complicated. Saleh and Gomaa proposes the feature description language to overcome such problem [27]. Its syntax looks similar to the C/C++ preprocessor but it supports separation of concerns by modularizing feature specific code in a separate file. So does our macro system in that expansion rules are maintains in a separate file in order to prevent the growth of complexity.

Sunkle et al. proposed features as first-class entities [28] but our approach focuses on not features but relations between features and assets so that a user's feature selection can pull the trigger at assembling, adapting and integrating core assets. Similarly, the AHEAD tools suite and FeatureHouse provide application generators but they treat a feature as a program increment or a program delta at the code level [23, 2]. These tools assume that there is a one-to-one mapping between conceptual *features* in feature modeling and concrete *features* in code. Therefore, in their approaches, the concepts at the requirement level and their realization at the code level should be similar while our approach studies features mainly in the requirement level and they are incrementally realized as assets during design and implementation phase. Therefore, the connection between mechanisms and each phase of the development process is of importance in our approach. This paper can be considered as an effort to make such connection explicit by providing relevant language supports.

The interpreter family example in this paper was originally designed to capture the so-called expression problem which describes the difficulty of two dimensional extensibility [13]. Variants of this problem have been popularly used to demonstrate the expressive power of feature-oriented programming. For example, as a variant, Lopez-Herrejon et al. defined *the Expression Product-Line (EPL)* to perform comparative study on feature modulization [23]. The aspect of generating different product-lines in the context of the expression problem has also been studied in the work on Origami [9]. In an Origami matrix, orthogonal features are described as either rows or columns and they are synthesized through a series of matrix transformation to generate a product-line. Since Origami is not implementation-specific, we believe that their matrix transformation (i.e., folding a matrix) can be implemented using our language features in a way that rows and columns are represented by our extensible module language and extensible first-class cases and then our macro system performs a matrix transformation by assembling them.

Recently, Apel et al. discussed feature (de)composition in the context of functional programming [1]. However, their interest lies in the problem of cross cutting in functional programming while we are interested in applying functional programming techniques (e.g., higher-order modules and type-safe extensible cases) into feature composition.

5. CONCLUSIONS AND OUTLOOK

Our work shows that advanced programming language technology such as extensible cases and parameterized modules are helpful when we express and implement variations identified by product line analysis. Furthermore, our macro system simplifies feature selection-based composition. Although each of these mechanisms alone may make feature-oriented development more convenient, building a family of products becomes easier when all mechanisms are available.

We are continuing this work in several ways. First, we plan to integrate a feature modeling tool with our work. Since our expansion rules do not support any specification of feature relationships (i.e., mutually exclusive or required relations), the **MLPolyR** compiler cannot detect any invalid feature sets. We leave such validation to feature modeling tools which provide various diagnoses on feature models. Our goal is to let a front-end modeling tool generate valid expansion rules. Then, application engineering would only require feature selection.

We will also continue to improve architectural expressiveness of **MLPolyR** to enable it to describe various view-

points. For example, FORM supports three viewpoints (i.e., subsystem, process and module) [18]. So far only modules have been the focus in our architecture models. Note that our programming styles rely on the fact that there are two kinds of variations: architectural variations and component-level variations. We conjecture that different architecture styles (i.e., layered or blackboard style [16]) may require different programming models. In this line of research, we are investigating the usage of more expressive architecture description languages such as ArchJava which provides more explicit notations for dynamic configuration of product lines [26].

6. REFERENCES

[1] S. Apel, C. Kästner, A. Größlinger, and C. Lengauer. Feature (de)composition in functional programming. In *Proceedings of the 8th International Conference on Software Composition (SC)*, pages 9–26, July 2009.

[2] S. Apel, C. Kästner, and C. Lengauer. FeatureHouse: Language-independent, automated software composition. In *Proceedings of the 31th International Conference on Software Engineering*, 2009.

[3] S. Apel, M. Kuhlemann, and T. Leich. Generic Feature Modules: Two-Staged Program Customization. In *Proceedings of International Conference on Software and Data Technologies*, pages 127–132, Sept. 2006.

[4] S. Apel, T. Leich, M. RosenmÃijller, and G. Saake. Featurec++: On the symbiosis of feature-oriented and aspect-oriented programming. In *Proceedings of the International Conference on Generative Programming and Component Engineering*, pages 125–140, 2005.

[5] A. W. Appel and D. B. MacQueen. Standard ML of New Jersey. In *Proceedings of the third International Symp. on Prog. Lang. Implementation and Logic Programming*, pages 1–13, New York, Aug. 1991.

[6] I. Aracic, V. Gasiunas, M. Mezini, and K. Ostermann. *An Overview of CaesarJ*. Lecture Notes in Computer Science : Transactions on Aspect-Oriented Software Development I, 2006.

[7] AspectJ. http://www.eclipse.org/aspectj/, 2008.

[8] D. Batory. Feature-oriented programming and the ahead tool suite. In *Proceedings of the International Conference on Software Engineering*, 2004.

[9] D. Batory, R. E. Lopez-Herrejon, and J.-P. Martin. Generating product-lines of product-families. In *Proceedings of the 17th IEEE International Conference on Automated Software Engineering*, page 81, 2002.

[10] M. Blume, U. A. Acar, and W. Chae. Extensible programming with first-class cases. In *Proceedings of the International Conference of Functional Programming*, pages 239–250, 2006.

[11] M. Blume, U. A. Acar, and W. Chae. Exception handlers as extensible cases. In *Proceedings of the ASIAN Symposium on Programming Languages and Systems*, 2008.

[12] W. Chae and M. Blume. Building a family of compilers. In *Proceedings of the 12th International Software Product Line Conference*, 2008.

[13] W. Chae and M. Blume. An evaluation framework for product line implementation. Technical Report TTIC-TR-2009, TTI at Chicago, 2009.

[14] H. Cho, K. Lee, and K. C. Kang. Feature relation and dependency management: An aspect-oriented approach. In *Proceedings of Software Product Line Conference*, 2008.

[15] K. Czarnecki and U. W. Eisenecker. *Generative programming: methods, tools, and applications*. ACM Press/Addison-Wesley Publishing Co., 2000.

[16] D. Garlan and M. Shaw. An introduction to software architecture. Technical report, Carnegie Mellon University, Pittsburgh, PA, USA, 1994.

[17] K. C. Kang, M. Kim, J. Lee, and B. Kim. Feature-oriented re-engineering of legacy systems into product line assets - a case study. In *Proceedings of the Software Product Line Conference*, pages 45–56, 2005.

[18] K. C. Kang, S. Kim, J. Lee, K. Kim, E. Shin, and M. Huh. Form: A feature-oriented reuse method with domain-specific reference architectures. *Ann. Softw. Eng.*, 5:143–168, 1998.

[19] K. C. Kang, J. Lee, and P. Donohoe. Feature-oriented product line engineering. *IEEE Softw.*, 19(4):58–65, 2002.

[20] C. Kästner and S. Apel. Integrating compositional and annotative approaches for product line engineering. In *Proceedings of the GPCE Workshop on Modularization, Composition and Generative Techniques for Product Line Engineering (McGPLE)*, pages 35–40, Oct. 2008.

[21] K. Lee, K. C. Kang, M. Kim, and S. Park. Combining feature-oriented analysis and aspect-oriented programming for product line asset development. In *Proceedings of the 10th International on Software Product Line Conference*, pages 103–112, 2006.

[22] K. Lee, K. C. Kang, and J. Lee. Concepts and guidelines of feature modeling for product line software engineering. In *Proceedings of the 7th International Conference on Software Reuse*, pages 62–77, 2002.

[23] R. E. Lopez-herrejon, D. Batory, and W. Cook. Evaluating support for features in advanced modularization technologies. In *European Conference on Object-Oriented Programming*, 2005.

[24] R. Milner. A theory of type polymorphism in programming. *J. Comput. Syst. Sci.*, 17(3):348–375, 1978.

[25] N. Noda and T. Kishi. Aspect-oriented modeling for variability management. In *Proceedings of the International Software Product Line Conference*, 2008.

[26] S. Pavel, J. Noyé, and J.-C. Royer. Dynamic configuration of software product lines in archjava. In *Proceedings of Software Product Line Conference*, pages 90–109, 2004.

[27] M. Saleh and H. Gomaa. Separation of concerns in software product line engineering. *SIGSOFT Softw. Eng. Notes*, 30(4):1–5, 2005.

[28] S. Sunkle, M. Rosenmüller, N. Siegmund, S. S. ur Rahman, G. Saake, and S. Apel. Features as first-class entities - toward a better representation of features. In *Proceedings of the GPCE Workshop on Modularization, Composition, and Generative Techniques for Product Line Engineering (McGPLE)*, pages 27–34, Oct 2008.

[29] P. Wadler. The expression problem, Dec. 1998. Email to the Java Genericity mailing list.

Feature-Oriented Programming with Ruby

Sebastian Günther and Sagar Sunkle
Faculty of Computer Science
University of Magdeburg
{sebastian.guenther|sagar.sunkle}@ovgu.de

ABSTRACT

Features identify core characteristics of software in order to produce families of programs. Through configuration, different variants of a program can be composed. Our approach is to design features as first-class entities of a language. With this approach, features can be constructed and returned by methods, stored in variables and used in many expressions of the language. This paper introduces rbFeatures, an implementation of first-class features in the dynamic programming language Ruby. Our goal is to show how such a language extension works with respect to its dynamic host language and the applicability of our results. In particular, we present a step-by-step walkthrough how to use rbFeatures in order to implement known case-studies like the Graph Product Line or the Expression Product Line. Since we created a pure Ruby language extension, rbFeatures can be used with any existing programs and in any virtual machine implementing Ruby.

Categories and Subject Descriptors: D.2.2 [**Software**]: Software Engineering - *Design Tools and Techniques*; D.3.3 [**Software**]: Programming Languages - *Language Constructs and Features*

General Terms: Languages

Keywords: Feature-Oriented Programming, Domain-Specific Languages, Dynamic Programming Languages

1. INTRODUCTION

Software has an inherent complexity. Since the advent of software engineering with the Nato conference in 1968, the question of how to cleanly modularize software into its various concerns is an ongoing question [1]. Today's challenges involve multiple requirements and different domains that software must consider. However, the tyranny of the dominant decomposition forces developers to decompose software along one dimension only [23]. This leads to several software defects, such as tangled and bloated code [13] and structural mismatches of requirements and programs because they can not be mapped one to one. Solution suggested to the code-tangling problems are concepts like Copliens ideas on multi-paradigm design [5], Kiczales et al. about Aspect-Oriented Programming [13] and Prehofer's Feature-Oriented Programming [20].This paper focuses on Feature-Oriented Programming, or short FOP.

Features are characteristics of software that distinguish members of a so called program family [4]. Program families are comparable with Software Product Line (SPL). An SPL is a set or related programs with different characteristics where the features reflect the requirements of stakeholders [11]. The challenge which SPL engineering addresses is to structure valuable production assets in a meaningful way to support productivity and reusability [6]. Withey further defines product lines as a "sharing a common, managed set of features" [25].

Features can be realized with very different approaches: mixin-layers [21], AHEAD-refinements [4], and aspectual feature modules [1] to name a few. Approaches can be differentiated [12] into compositional (e.g., features are added as refinements to a base program [4]) and annotative (e.g., features are implemented as annotations inside the source code [12]).

Because of the several limitations and drawbacks that named FOP approaches have [18, 12, 22], we are actively suggesting new concepts that close the gap between the conceptual and the implementation view of features. One such suggestion was to implement features as first-class entities in a host language. We implemented features as first-class entities with the static programming language Java as the host language called *FeatureJ*[1] [22]. First-class entities are characterized by the following abilities: They can be instantiated at run-time, stored in variables, used as parameters to methods or being returned from them, and also used in various statements and expressions. The gap is closed if we extend a host language with first-class features [22]. In such case features are language-level entities used actively in programming while retaining their status as a high-level modularization concept.

This paper supports our ideas by providing Feature-Oriented Programming (FOP) for the dynamic programming language Ruby. We named our FOP implementation *rbFeatures*[2]. rbFeatures is a language extension for Ruby that provides features as language entities in Ruby. Our motiva-

FOSD'09, October 6, 2009, Denver, Colorado, USA.

[1]http://firstclassfeatures.org/FeatureJ.htm
[2]http://firstclassfeatures.org/rbFeatures.htm

tion is twofold. At first, we want to create another implementation of features as first-class entities to better compare with the Java version and other approaches. We await to gain more insight into language extensions that can be realized with respect to static and dynamic programming languages. The second part is applicability of the result. Since rbFeatures is implemented in pure Ruby, using no external libraries, it is widely usable with both different virtual machines and types of programs. Such discussions however will be deferred to the later part of the paper - the focus is to explain how to use rbFeatures and how features are realized as first-class entities.

The remainder of this paper is structured as follows. The next section will further explain feature-terminology and characteristics. Section 3 introduces the core concepts of the Ruby language. Section 4 explains rbFeatures in detail with a walkthrough by defining an example. We further explain other examples in Section 5. The last two sections discuss our experiences and related work.

2. FEATURE-ORIENTED PROGRAMMING

Modern FOP methods should combine both the conceptual view and the implementation view on features. Considering our earlier proposal to close the gap between the conceptual and implementation views on features [22], we want to elaborate a refined terminology and show properties of first-class features.

2.1 Feature Terminology

We observe that features try to tackle two main problems of software engineering. On the one hand to capture the conceptual higher-level view in order to abstract and configure core functionality, and on the other hand to provide a low-level implementation view for identification and composition of feature-relevant parts of a program.

From a conceptual viewpoint, a feature is an abstract entity which expresses a concern. Concerns can be general requirements of stakeholders [11] or increments in functionality [12]. From the functionality viewpoint, features describe modularized core functionality that can be used to distinguish members of a program family [4]. The functionality viewpoint is taken from here on.

A feature plays an important role in the analysis, design, and implementation phases of software development. In the first part of the development, *conceptual features* describe in natural language, the name, intent, scope, and functions that a feature provides to a program. Conceptual features are thus an additional unit to express the programs decomposition - providing some remedy to the tyranny of the dominant decomposition problem. In the implementation phase, *concrete features* are constructed. Concrete features are the implementation of features inside the program. Often, these concrete features are seen as refinements that add code to a certain base program, e.g. as shown in typical FOP case studies like the graph product line [14] or in the expression product line [15].

This describes our terminology in the remainder of the paper. If not stated otherwise, we use the word feature to denote a concrete feature.

2.2 Feature Properties

Features have properties needed to define their characteristics and role they have in programming. We see properties as defining steps when using FOP. We studied the suggested properties of Lopez-Herrejon [15]. From the viewpoint of dynamic languages, the characteristics *Separate Compilation* and *Static Typing* have no meaning because Ruby is interpreted and typeless. We abstract and generalize the other characteristics as followed (we name the grouping at the end of each property in *italics text*).

- **Naming** Conceptual features are means to abstract the core functionality. This functionality is given a unique name and intent. The concrete feature should also have the same name, while presenting its intent with the following *identification* and *composition* mechanisms. (*Cohesion*)
- **Identification** To form concrete features, one must first identify what parts of the program belong to feature. Those parts can be coarse or fine grained. Coarse-grained features can be thought of as stand-alone parts of the program - they cleanly integrate with the base program via defined interfaces or by adding new classes. On the contrary, fine-grained features impact various parts of the source code: extending code lines around existing blocks of code, changing parameters of methods, or adding single variables at arbitrary places. (*Deltas*)
- **Expressing** A concrete feature needs not only to specify what parts of the program belong to it, but also how these parts are added to the program. Such expressions must work on the same abstraction level as the programs language. (*Deltas*)
- **Composition** Once all feature related code is identified and expressed, the composition is the process to configure and build a variant. It is desirable that the composition-order of features imposes no constraint to compose a working program. (*Flexible Composition*, *Flexible Order*, *Closure under Composition*)

3. RUBY PROGRAMMING LANGUAGE

Ruby is a completely object-oriented dynamic programming language. Its inventor, Yukihiro Matsumoto, took the best concepts of his favorite programming languages and put them inside one language. His foremost motivation was to make programming faster and easier [10].

Ruby has many capabilities for flexible extension and modification of the source code. Naturally, this makes Ruby a good vehicle for FOP. This section introduces Ruby with the most important concepts which play a role in rbFeatures, so that readers yet unfamiliar with Ruby can better follow the ideas proposed later. All material in this section stems from [10] and [24].

3.1 Class Model

Five classes are the foundation of Ruby. At the root lies *BasicObject*. It defines just a handful of methods needed for creating objects from it and is typically used to create objects with minimal behavior. *Object* is the superclass from which all other classes and modules inherit. However, most of its functionality (like to copy, freeze, marshal and print objects) is actually derived from *Kernel*. Another important class is *Module*, which mainly provides reflection mechanisms, like getting methods and variables, and metaprogramming capabilities, e.g. to change module and class definitions. Finally, *Class* defines methods to create instances of classes.

3.2 Core Objects

We discuss the objects of the classes Proc, Method, Class, and Module. The core objects play a fundamental part in rbFeatures. Most of these objects should be familiar to readers experienced with object-oriented programming. However, Ruby's dynamic nature makes following objects more versatile compared to static languages.

- **Proc** A Proc is an anonymous block of code. Like other objects, Procs can either be created explicitly or referenced by a name or implicitly as an argument to a method. Procs allow two kind of usages: On the one hand they can reference variables in their creation scope as a closure[3], and on the other hand they can reference variables which at their creation do not exist. Procs are executed with the **call** method in any place.
- **Method** As a language entity, methods consist of the method's name, a set of (optional) parameters (which can have default values), and a body. Methods are realized inside modules and classes. There are two kinds of Method objects. The normal *Method* is bound to a certain object upon which it is defined. The *UnboundMethod* has no such object, and, for executing it, must be bound to an object of the same class in which the method was originally defined.
- **Class** A class consists of its name and its body. Each class definition creates an object of class Class. Classes can have different relationships. First, they can form a hierarchy of related classes via single sub-classing, inheriting all methods of their parents. Second, they can mix-in arbitrary modules.
- **Module** Modules have the same structure as classes do. Modules can not have subclassing like inheritance relationships - they can only mix-in other modules.

3.3 Classes instead of Types

Ruby does not have a static language-like type hierarchy for its classes and objects. The now deprecated[4] method *Object.type* would just return the name of the class the object corresponds to. Instead of a type, the classes inside Ruby are identified according to the methods they provide. An object of any class can be used in all contexts as long as it responds to the given method calls. In the Ruby community, this is called duck typing: "If an object walks like a duck and talks like a duck, then the interpreter is happy to treat it as if it were a duck"[24]. One usage of duck typing is to swap the used objects at runtime, e.g. to provide more performance for data storage.

With this, we finish our introduction of basic concepts. Many more details can be found in [10] and [24]. The next section explains rbFeatures in detail.

4. RBFEATURES

rbFeatures is a language extension to Ruby for realizing Feature-Oriented Programming. Language extension here means two things: At first, new entities and operations are added that represent features at the source-code level, and second that the feature properties defined in section 2.2 to name, identify, express and compose features are supported.

[3]Closures stems from functional programming and capture values and variables in the context they are defined in.

[4]This method was declared deprecated in Ruby 1.8, and has vanished from the current 1.9 release.

To convey our ideas, we choose to present a side-by-side rbFeatures walkthrough. We start with introducing a running example, present the three steps of naming and identifying features, composing a product line and using the product line. Each section motivates an example and explains rbFeatures details. At the end, we summarize the rbFeatures workflow.

For clarity, we introduce two new notations: Slanted text for FEATURES as defined in the product line, and verbatim text for `language entities` which are part of rbFeatures and the developed product line.

4.1 Example: Expression Product Line

The Expression Product Line (EPL) is a common problem used in programming language design as exemplified by Lopez et. al. in [15]. EPL considers the case of different types of numbers, expressions and operations, which are used to compare design and synthesis of variants [15]. The product line is presented as a feature diagram in ►Figure 1. We see that two different NUMBERS, LIT (literal) and NEG (negative literal) exist. The numbers are usable in ADD (addition) and SUB (subtraction) EXPRESSIONS. The OPERATIONS imposed upon an expression are PRINT for showing the string containing the expression and EVAL for evaluating the expression. As can be seen from ►Figure 1, we define the product line with many of its features as being optional, so no strict rules governing the composition need to be considered.

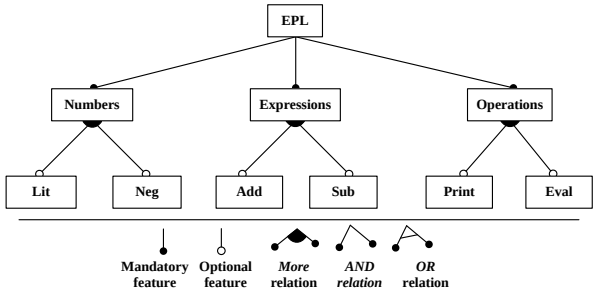


Figure 1: EPL Feature Diagram

4.2 Implementing the Expression Product Line

In three steps we will show how to implement features, define the product line and finally compose and use its variants. Ruby does not impose a fixed structure where expressions must be defined. Programmers can split module and class definitions in separate files, put it all in one file or define it on-the-fly in a shell session using IRB (Interactive Ruby). The original source code for the EPL is contained in one file only.

Step 1: Defining Features

The first step is to define the features. This is done with a very concise notation: objects of class **Class** simply include the `Feature` module. To define the root features, one must write the following (cf. ►Figure 2, left part).

Afterwards, concrete numbers and operations are defined. These features also contain methods themselves. We implement LIT with the following code (cf. ►Figure 2, right part). In line 1 we declare the class `Lit` to be a subclass

```
1  class Numbers
2    is Feature
3  end
4
5  class Expressions
6    is Feature
7  end
8
9  class Operations
10   is Feature
11 end
```

```
1 class Lit < Numbers
2   def initialize(val)
3     @value = val
4   end
5 end
```

Figure 2: Defining basic Features

of `Numbers`. With subclassing, we express the natural tree-structure of the product line. The subclasses here inherit the methods of module `Feature` from their parents. We then define in line 3 the `initialize` method which receives the initial value for `Lit`. We implement `Neg` likewise.

This is followed by defining `Add` with the following code (cf. ►Figure 3). We use again subclassing to define the relationship to `Operations}`, and define the `initialize` method to get two values (which can be `Add`, `Sub`, `Neg` and `Lit` values). Again, `Sub` is implemented likewise.

```
1 class Add < Operations
2   def initialize(left, right)
3     @left = left
4     @right = right
5   end
6 end
```

Figure 3: Defining the Add Operation

Having defined all `Numbers` and `Expressions`, we now define the `Print` feature (cf. ►Figure 4). We then add `print` methods to `Lit` and `Add`. Line 5 shows a *feature containment*.

```
1  class Print < Operations
2  end
3
4  class Lit
5    Print.code do
6      def print
7        Kernel.print @value
8      end
9    end
10 end
```

Figure 4: Defining Print Feature and Operations

The first part, just `Print` in this case, is the *containment condition*. A containment condition is an arbitrary expression combining features with operations. The second part is the *containment body*. The body is surrounded by a **do...end** block, and contains in this case a method definition for `print`. The containment body can be any Ruby statement. Feature containments can be used to encapsulate coarse-grained modules, classes, and methods, as well as fine grained individual lines or even parts of a source code line. In total, the containment from line 5 to 9 expresses that only if `Print` is activated, then the `print` method is defined properly. Likewise, implementing the EVAL feature is expressed as a feature containment with `Eval` as the containment condition, and a declaration of the `eval` method (which returns the value of `Lit`) as the containment body.

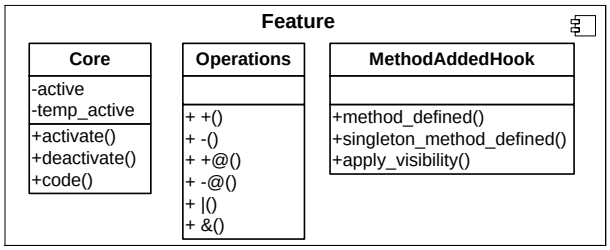


Figure 5: The Feature Module

The internals of features are as follows. rbFeatures defines the `Feature` module as a composition of three different modules, as seen in ►Figure 5. Each module encapsulates a set of operations needed for `Feature` to function properly:

- **Core** Provides the basic method to use a `Feature` as a configuration unit. We spoke of activating a feature - meaning that the internal class-variable `active` is set to true. The methods `activate` and `deactivate` change the activation status. The other operation is `code`, which was used in the example for defining the `print` method. The `code` receives a `Proc` object that is the containments body. Internally, `code` checks if the feature is activated - if yes, the body gets executed.
- **Operations** This module defines operations[5] which can be used inside the containment condition:
  - Plus (+) => Activation (single feature)
  - Minus (-) => Deactivation (single feature)
  - And (&) => Activation (all features)
  - Or (|) => Activation (at least one feature)

  Operations and features can be chained to arbitrary expressions. Conditions like "If feature A, B and C, but not D are activated", translate to a natural syntax `(A + B + C - D)`. Or "Feature A or B and C, but not D" translates to `((A | (B & C)) - D)`.
- **MethodAddedHook** This module defines a special hook. When methods are defines in an object, the private method *method_added* is called. We use this as a hook in the case of a feature containment defining a method. If the containments condition is not true, then the methods body will be replaced with a error message. This error message details which features activation status prohibits the method's definition. For example, if the `print` method is called on a `Lit`, but feature `Print` is not activated, the error message will be "*FeatureNotActivatedError: Feature Print is not activated*". This error message is computed dynamically and names the right-most feature inside a containment condition that has the wrong activation status. The `apply_visibility` method is needed to retain the visibility of the defined method in the case its body is overwritten.

[5]Concerning the method-declaration as shown in ►Figure 5: The methods containing an "@" symbol are unary operations defined on the object itself.

Step 2: Implementing the Product Line

Assume that all other features and the concrete semantics of each entity have been implemented. We now need to implement the Product Line. Currently, we use no special object, but a native `Proc` object. This allows two composition approaches: To manually surround all code in a `lambda do; ...; end` block (cf. ►Figure 6, line 1 to 5) or to join the content of all source files and define a lambda from them. To stay at a higher abstraction level, we usually name this proc as the product line it represents.

```
1 EPL = lambda do
2 class Lit < Numbers
3   def initialize(value)
4 ...
5 end
6
7 FeatureResolver.initialize EPL
```

Figure 6: Defining and Initializing the EPL

Having defined the product line, the user then directly interacts with a module called `FeatureResolver`. This module handles initialization and resetting of the product line. As shown in ►Figure 7, four methods are available. We explain in opposite direction. With `reset!`, all classes which include the `Feature` module are deleted so that their initial definition can be restored. The `update` method is called whenever a feature changes its activation status. Finally, `init` and `register` are used during initialization.

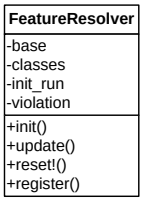


Figure 7: The Feature Resolver Module

The initialization composes the product line by evaluating the proc. Evaluating means that all code is executed once, and this leads to classes and modules defining the program. Classes including the `Feature` module register themselves with the `FeatureResolver`. A challenge is to regard classes or methods defined in feature containments. Most feature containments will not be executed because one feature may violate the containment condition. As mentioned before, rbFeatures informs the user if a method can not be called because of a certain feature violating the containment condition. Methods may be defined in containments which are violating a condition. How to define those methods properly? In the initialization phase, all containments are executed once - even if a violation occurred. This can lead to a method declaration, which is caught by the `MethodAddedHook`. If a violation occurred and the initialization phase is happening, then the original method body will be replaced with the error message. In the case of features defining functions themselves, their methods get deactivated with the same mechanism. Once the product line is composed, we can start using it actively.

Step 3: Composing and Using Variants

After initialization, the EPL is ready to use. The dynamic nature of Ruby allows two kinds of usages: static configuration of a variant before executing the program and dynamic configuration at runtime.

We start with static configuration. By defining the activation or deactivation of certain features, we define a variant. For example, we want to define a EPL with features Lit, Add, Sub and Print. This configuration can be represented as a class containing activation statements (cf. ►Figure 8).

```
1 class Variant1
2   Lit.activate
3   Add.activate
4   Sub.activate
5   Print.activate
6 end
```

Figure 8: Static Configuration of EPL

```
1  >> FeatureResolver.init ExpressionProductLine
2  => true
3  >> Add.activate
4  => :activated
5  >> Lit.activate
6  => :activated
7  >> Lit 1
8  => #<Lit:0xb7b35968 @value=1>
9  >> a = Add Lit(11), Lit(7)
10 => #<Add:0xb7c57544 @right=#<Lit:0xb7c57558
       @value=7>, @left#<Lit:0xb7c5756c @value=11>
11 >> a.print
12 FeatureNotActivatedError: Feature Print is not
       activated
13
14 >> Print.activate
15 => :activated
16 >> a.print
17 11+7=> nil
18 >> a.eval
19 FeatureNotActivatedError: Feature Eval is not
       activated
20 >> Eval.activate
21 => :activated
22 >> a.eval
23 => 18
24 >> Print.deactivate
25 => :deactivated
26 >> a.print
27 FeatureNotActivatedError: Feature Print is not
       activated
```

Figure 9: Session with EPL

The second usage kind is dynamic configuration. At runtime, we can activate and deactivate arbitrary features. Consider the following session with the interactive Ruby shell (►Figure 9). In the session, lines starting with "`>>`" denote input, and lines with "`=>`" denote output. In line 2 we load the `ExpressionProductLine` into the `FeatureResolver` module. Following lines 4 - 7 activate the features Add and Lit. We then create a single Lit object (line 8), and a compound Add object (line 10). Line 11 shows the return value of object creation - its in-memory representation. We then want to call `print` on the add statement (line 12), but get a `FeatureNotActivatedError` in return. After activation in line 15, we can print and see `11+7` in line 18. The same is shown for `eval` in lines 19-25. After that, we decide to de-

activate `Print` again by calling the same-named method in line 26. Successively, calls to `print` fail again.

4.3 Summary Workflow

Summarizing, using rbFeatures consists of the following steps. ►Figure 10 shows these steps graphically.

- Name and identification of the features
- Defining features via including the feature module and building subclass relationships
- Put all source code inside a Proc
- Initialize the proc to compose a variant
- Static or dynamic configuration of the variant

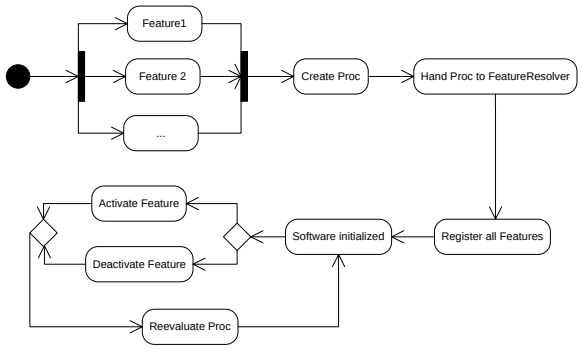


Figure 10: rbFeatures' basic Workflow

5. OTHER EXAMPLES

The EPL illustrates rbFeatures with a simple scenario. We also developed two other programs which are presented shortly. Our major motivation is to show that rbFeatures was successfully applied to two other product lines, with the second example having a graphical representation as well.

5.1 Graph Product Line

The Graph Product Line (GPL) describes a family of related program in the domain of graphs [14]. We see a graphical representation of the features and their relationships in ►Figure 11. We assume the reader understands this figure with respect to the legend in ►Figure 1.

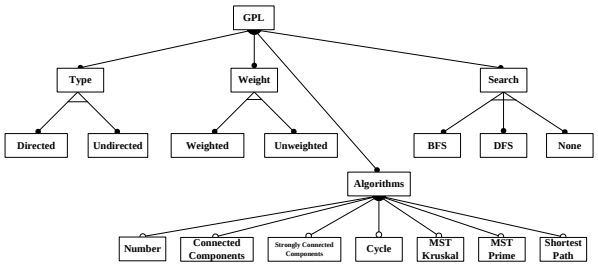


Figure 11: The Graph Product Line

Although heavyweight graph computations demand a solution using matrices, we implemented all entities of the domain, like nodes and edges, as objects. This was also reported in [14] as a better way to compose the program. Our approach was to first develop the whole program without thinking about features. Once all algorithms were implemented, we feature refactored the program. Refactoring means to define concrete features with the same name as shown on the feature diagrams, and then to identify and express those parts of the program which belong to a certain feature.

An example of the refactoring is shown in ►Figure 12. The feature Weight modifies the `Edge` class by forming containments around the `attr_accessor` (line 3) and by adding another line which deletes any given weights to edges so that the body is not changed (line 5).

```
1  class Edge
2    attr_reader :source, :sink
3    Weighted.code { attr_accessor :weight }
4    def initialize(params)
5      Weighted.code { @weight = params.delete :weight }
6      params.delete :weight if params.include? :weight
7      @source = params.first[0]
8      @sink = params.first[1]
9    end
10 end
```

Figure 12: Feature-Refactoring Changes in GPL

Feature containments for Type only wrapped existing lines. Finally, all Algorithms and Search features are put in containments. No further changes are required - even if features depend on other features. Consider the case of using Strongly Connected Components. It requires to use DFS. If a variant is created without activating DFS, then calling the method `strongly_connected_components` simply returns a "*FeatureNotActivatedError: Feature DFS is not activated*". Activating DFS remedies this situation.

5.2 Calculator Product Line

In the third example, we targeted an open source implementation of a simple graphical calculator. Feature refactoring changes to the source code were of medium nature, because the original program was very concise and used batch-like operation for method declaration. We also added a graphical configurator. At runtime, multiple instances of the Calculator, configured in different variants, can be created (►Figure 13).

6. COMPARISON AND DISCUSSION

In order to integrate rbFeatures into overall FOP research, we want to discuss a number of points. In our view, the most distinguishing point are to realize FOP as a pure and lightweight language extension, to use a dynamic programming language and finally the possibility to include modeling capabilities for further closing the gap between abstract and concrete features.

Language Extension

Many techniques have been introduced that implement FOP [2, 3, 17, 19]. Features are represented in terms of refinements, feature structure trees, more flexible containers than classes and interfaces, and hyperspaces. These approaches either impose modifications to the existing tools used in writing and generating the programs, like compiler extensions,

Figure 13: Configuring and Executing Variants of the Calculator Product Line

or use an external structure which maps features to an existing source code. rbFeatures has no such restrictions.

First, it is based on pure Ruby - no change to the interpreter and no additional library is needed. Programmers just need a virtual machine implementing Ruby 1.8.6 respective 1.9, such as MRI[6] written in C or JRuby[7] written in Java. The VM's capability may further broaden what programs can be written using rbFeatures. JRuby, which uses Java as its interpreter language, allows many cross-language developments, like calling Java code from inside Ruby or vice versa [9]. As applications written in JRuby can access native Java libraries, those libraries become feasible for FOP using rbFeatures.

Second, rbFeatures operates on the same abstraction level as the program - the language level. Developers directly include feature containments at the place in the program where code belonging to a feature is expressed. They do not need to switch to another representation, possible using another language or syntax. In terms of the feature properties in section 2.2, the tasks of feature identification and expression become the same in rbFeatures. Further more, the amount of changes to a program without features is very minimal. Typical changes just introduce containments with a condition around an existing block of code. At least with the presented case studies, we seldom had a case where we need to rewrite the code. ►Table 1 shows what changes occurred in the show examples when rbFeatures was used. Column 2 and 3 lists the loc for the normal and feature-based version of the program. Column 4 shows the total LOC added, followed by percental increase. We see that smaller programs tend to have a comparatively big amount of changes when using rbFeatures. For programs with a large LOC, the lines added from using rbFeatures are relatively small. Also, the number of methods is an indicator for changes. For example, the changes in EPL were mostly the containments `do...end` block around the methods.

Dynamic Language vs. Static Language

Techniques implemented in statically typed object-oriented programming languages have dominated research in FOP. Most of the aforementioned techniques are implemented in more conventional programming languages like Java and C++, which are usually static. On the contrary, Ruby's dynamic typing and interpreted execution enables metaprogramming mechanisms which strengthens implementing language extensions. Proc objects defined at a certain place in the program can either be called at this place or called in another context. And since the variables contained inside a proc can either reference existing ones or yet to be defined ones allows very flexible composition. But this flexibility comes at a price. First, static languages are more efficient then interpreted languages in terms of runtime and performance. Second, typeless languages can introduce bugs to programs that are only countered with testing. We used a fully test-driven approach to rbFeatures and tested all mentioned properties in different contexts.

[6]http://www.ruby-lang.org/en/
[7]http://jruby.codehaus.org/

PL	Normal	Feature	#LOC	%LOC	Methods
EPL	74	97	23	31%	10
CPL	70	89	19	27%	4
GPL	291	323	32	11%	9

Table 1: Comparing LOC for rbFeatures Programs

Including Feature Modeling Capabilities

rbFeatures can be seen as a so-called Domain Specific Language (DSL). It uses suitable notation and abstraction to represent domain-knowledge and concepts in a precise form [8]. For rbFeatures, this domain is feature-oriented programming. But, we can extend rbFeatures with concepts from textual feature modeling languages to allow expressing both conceptual features and concrete features.

The array of feature modeling languages is vast - we want to give two examples here. Deursen and Klint propose a feature description language in [7] in which they consider automated manipulation of feature descriptions. Feature composition is achieved by translating the textual feature descriptions to unified modeling language models. Similarly, Loughran et al. [16] present the variability modeling language (VML). VML supports first-class representation of architectural variabilities. VML consists of explicit references to variation points and composition of both fine and coarse-grained variabilities. With this capability, VML resembles more an architectural description language.

Expressing conceptual features in rbFeatures requires a language extension which uses similar metaprogramming concepts as shown before. We would need a first-class representation of a product line, product variant, and constraints regarding legal combinations of features. This is future work.

7. SUMMARY

rbFeatures is a pure language extension to Ruby in order to enable feature-oriented programming. We introduced the Ruby language, and presented a step-by-step walkthrough how to use rbFeatures and how the extension works internally. By discussing the main cases studies of FOP, namely the Graph Product Line and the Expression Product Line, we enable the direct comparison of rbFeatures with other approaches.

Defining features as first-class entities leads to full usability of features in all parts of a program. Features can be stored in variables, returned from methods and extended like any other object. Only minimal changes are required to make a program feature-oriented. Because of the lightweight implementation using only core Ruby mechanisms, rbFea-

tures is useable with all other virtual machines.

We want to extend rbFeatures in several ways. Additional to known FOP case studies, we want to use rbFeatures also in the context of web applications written with the Rails framework. Furthermore, we want to combine rbFeatures with a already implemented software product line configuration language for abstract modeling and concrete implementation at the same abstraction level.

Acknowledgements

We thank Christian Kästner for his comments on an earlier draft of this paper.

8. REFERENCES

[1] S. Apel. The role of features and aspects in software development. PhD Thesis, Otto-von-Guericke-Universität Magdeburg, 2007.

[2] S. Apel, T. Leich, M. Rosenmüller, and G. Saake. FeatureC++: On the Symbiosis of Feature-Oriented and Aspect-Oriented Programming. In *Proceedings of the 4th International Conference on Generative Programming and Component Engineering*, Lecture Notes in Computer Science, Springer, Heidelberg, 3676:125–140, 2005.

[3] D. Batory. Feature-Oriented Programming and the AHEAD Tool Suite. In *Proceedings of the 26th International Conference on Software Engineering*. IEEE Computer Society, pages 702–703, 2004.

[4] D. Batory, J. N. Sarvela, and A. Rauschmayer. Scaling step-wise refinement. *IEEE Transactions on Software Engineering*, 30:355–371, 2004.

[5] J. O. Coplien. Multi-paradigm design. PhD Thesis, Vrije Universiteit Brussels, 2000.

[6] K. Czarnecki and U. W. Eisenecker. *Generative programming: methods, tools, and applications*. Addison-Wesley, Boston, San Francisco et. al., 2000.

[7] A. v. Deursen and P. Klint. Domain-specific language design requires feature descriptions. *Journal of Computing and Information Technology*, 10(6):1–17, 2002.

[8] A. v. Deursen, P. Klint, and J. Visser. Domain-specific languages: an annotated bibliography. *SIGPLAN Not.*, 35(6):26–36, 2000.

[9] J. Edelson and H. Liu. *JRuby Cookbook*. O'Reilly Media Inc., Sebastopol, 2008.

[10] D. Flanagan and Y. Matsumoto. *The Ruby Programming Language*. O-Reilly Media Inc., Sebastopol, 2008.

[11] K. Kang, S. Cohen, J. Hess, W. Novak, and A. Peterson. Feature-oriented domain analysis (foda) feasibility study. *Technical report CMU/SEI-90-TR-021*, Carnegie Mellon University, 1990.

[12] C. Kästner, S. Apel, and M. Kuhlemann. Granularity in software product lines. In *Proceedings of the 30th International Conference on Software Engineering (ICSE)*, ACM, New York, pages 311–320, 2008.

[13] G. Kiczales, J. Lamping, A. Menhdhekar, C. Maeda, C. Lopes, J.-M. Loingtier, and J. Irwin. Aspect-oriented programming. In *Proceedings of the 11th European Conference on Object-Oriented Programming (ECOOP)*, Lecture Notes in Computer Science, Springer, Berlin, Heidelberg et. al, 1241:220–242, 1997.

[14] R. E. Lopez-Herrejon and D. Batory. A standard problem for evaluating product-line methodologies. In *Proceedings of the 3rd International Conference on Generative and Component-Based Software Engineering (GCSE)*, Lecture Notes in Computer Science, Springer, Berlin, Heidelberg, 2186:10–24, 2001

[15] R. E. Lopez-Herrejon, D. Batory, and W. Cook. Evaluating support for features in advanced modularization techniques. In *Proceedings of the 19th European Conference on Object-Oriented Programming (ECOOP)*, Lecture Notes in Computer Science, Springer, Berlin, Heidelberg, 3586:1–37, 2005.

[16] N. Loughran, P. Sánchez, A. Garcia, and L. Fuentes. Language support for managing variability in architectural models. In *Proceedings of the 7th International Symposium on Software Composition (SC)*, Lecture Notes in Computer Science, Springer, Berlin, Heidelberg 4954:36–51, 2008.

[17] S. McDirmid, M. Flatt, and W. C. Hsieh. Jiazzi: New-Age Components for Old-Fashioned Java. *ACM Press*, pages 211–222, 2001.

[18] M. Mezini and K. Ostermann. Variability management with feature-oriented programming and aspects. *ACM SIGSOFT Software Engineering Notes*, 29(6):127–136, 2004.

[19] H. Ossher and P. Tarr. Hyper/J: Multi-Dimensional Separation of Concerns for Java. In *Proceedings of the 23rd International Conference on Software Engineering (ICSE)*, IEEE Computer Society, pages 729–730, 2001.

[20] C. Prehofer. Feature-oriented programming: A fresh look at objects. In *Proceedings of the 11th European Conference on Object-Oriented Programming (ECOOP)*, Lecture Notes in Computer Science, Springer, Berlin, Heidelberg, 1241:419–443, 1997.

[21] Y. Smaragdakis and D. Batory. Mixin layers: An object-oriented implementation technique for refinements and collaboration-based designs. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 11(2):215–255, 2002.

[22] S. Sunkle, M. Rosenmüller, N. Siegmund, S. S. Ur Rahman, G. Saake, and S. Apel. Features as first-class entities - toward a better representation of features. In *Workshop on Modularization, Composition, and Generative Techniques for Product Line Engineering*, pages 27–34, 2008.

[23] P. Tarr, H. Ossher, W. Harrison, and S. M. J. Sutton. N degrees of separation: Multi-dimensional separation of concerns. In *Proceedings of the 21st International Conference on Software Engineering (ICSE)*, ACM, pages 107–119, 1999.

[24] D. Thomas, C. Fowler, and A. Hunt. *Programming Ruby 1.9 - The Pragmatic Programmers' Guide*. The Pragmatic Bookshelf, Raleigh, 2009.

[25] J. Withey. Investment analysis of software assets for product lines. *Technical Report CMU/SEI96-TR-010*, Carnegie Mellon University, 1996.

Remodularizing Java Programs for Comprehension of Features

Andrzej Olszak and Bo Nørregaard Jørgensen
The Maersk Mc-Kinney Moller Institute
University of Southern Denmark
Campusvej 55, 5230 Odense M, Denmark
{ao, bnj}@mmmi.sdu.dk

ABSTRACT
Feature-oriented decomposition of software is known to improve a programmer's ability to understand and modify software during maintenance tasks. However, it is difficult to take advantage of this fact in case of object-oriented software due to lack of appropriate feature modularization mechanisms. In absence of these mechanisms, feature implementations tend to be scattered and tangled in terms of object-oriented abstractions, making the code implementing features difficult to locate and comprehend. In this paper we present a semi-automatic method for feature-oriented remodularization of Java programs. Our method uses execution traces to locate implementations of features, and Java packages to establish explicit feature modules. To evaluate usefulness of the approach, we present a case study where we apply our method to two real-world software systems. The obtained results indicate a significant improvement of feature representation in both programs, and confirm the low level of manual effort required by the proposed remodularization method.

Categories and Subject Descriptors
D.2.7 [**Software Engineering**]: Distribution, Maintenance, and Enhancement – *restructuring, reverse engineering, and reengineering.*

General Terms
Design

Keywords
Features, feature location, remodularization.

1. INTRODUCTION
Program comprehension was reported to consume more than half of all resources during software maintenance [1]. An important part of the overall comprehension effort is spent on relating program features to source code [2]. A *feature*, being a cohesive set of functionality [2], serves software users as a means for formulating their requirements, change requests, and error reports. In turn, developers need to relate the descriptions obtained from the users to appropriate fragments of a program's source code in order to carry out the requested modifications. Thus, the easier it is to understand correspondence between features and their implementations, the easier it is to modify a program's functionality [3].

In object-oriented software, mapping between features and source code is often not explicit. This is due to the fact that decompositions of object-oriented programs tend to follow architectural styles based on layered arrangement of code [4]. Features are not represented explicitly in such decompositions. Instead, implementations of individual program features crosscut multiple modules and multiple architectural layers. This scattering makes it a difficult and tedious task to identify the classes and methods that implement a given feature. As a result, programmers need to use additional effort to understand how features are implemented, and how their implementations relate to each other.

Comprehension of features in existing object-oriented software can be enhanced by visualizing the mapping between features and code [5][6]. Furthermore, to address the problems of feature implementations' scattering and tangling, a one-to-one correspondence between a program's functionality and its static structure is needed. Modularizing software according to its features creates such a correspondence by encapsulating feature implementations in terms of distinct modules. Feature-oriented decomposition is expected to enable the three main outcomes of modularity [7]: feature-wise program understanding, feature-wise division of work [8], and confinement of change propagation to boundaries of features' implementations.

However, achieving the aforementioned comprehension benefits in existing object-oriented programs is difficult due to the complexity of the remodularization process. Insufficient level of automation of the time-consuming and error-prone restructuring activities was found to be a significant barrier for scalability of the existing remodularization approaches [9]. This suggests that if an approach for feature-oriented remodularization is to be adopted in real-world software projects it needs to require very little manual work. Ideally, it should be possible to remodularize software in an on-demand fashion [10], so that programmers could switch to the decomposition best suited for accomplishing a task at hand in a completely automatic manner.

Finally, during remodularization of existing programs human-understandable concepts present in the original decomposition should be taken into account and preserved. Such concepts reflect a program's problem domain, and are usually implemented in the code in form of a *domain model* [11]. For the programmers, who are familiar with the original codebase, it is crucial to preserve these concepts in order to increase their understanding of the new decomposition [12].

In this paper we present an approach for semi-automatic remodularization of existing Java programs towards feature-

FOSD'09, October 6, 2009 Denver, Colorado, USA

oriented decomposition. Our approach represents features in terms of explicit feature modules. To locate source code fragments that implement program features we use execution tracing based on a proposed notion of feature entry points. For the purpose of creating feature modules we use Java packages. Our approach progresses the state of the art in the area of automated refactoring of object-oriented software towards explicit representation of features. Furthermore, our approach points out the important problem of preserving concepts present in the original program throughout the remodularization process.

To evaluate our approach we have conducted a case study on two real-world software systems: JHotDraw and BlueJ. The obtained results show that we have substantially improved the representation of features in the created modularizations.

The rest of this paper is organized as follows: Section 2 provides an overview of our remodularization approach. Section 3 describes the implementation details of the approach. Section 4 presents two case studies, and discusses the results. Section 5 presents related work. Finally, Section 6 concludes the paper.

2. OVERVIEW

Our remodularization approach establishes a feature-oriented decomposition for a Java program. The approach was designed to be used for reengineering of existing software, and to require as little manual effort as possible.

The idea behind our approach is to create an explicit feature representation for a Java program by grouping classes that implement distinct program features. Our remodularization approach consists of two subsequent steps: *feature location* and *feature representation*. The semi-automatic feature location procedure identifies methods that participate in implementations of program features. Then, a feature representation is established by assigning classes to their corresponding feature modules. The remainder of this section provides a conceptual overview of these two steps, whereas their details are given in the next section.

2.1 Feature Location

Feature location is concerned with identifying source code that contributes to implementations of program features [13][14]. Our feature location procedure uses *feature entry points* to recognize execution of concrete features at a program's runtime.

We propose the notion of a feature entry point. By a feature entry point we understand the first method through which a thread of execution enters when a user interacts with a program feature. Every method called inside the control flow of a feature entry point is assumed to belong to the implementation of that feature entry point's feature.

In order to declare a method as being a feature entry point, a programmer needs to manually tag the candidate method using a Java annotation. The reasons for using annotations for this purpose are that they do not impact a program's correctness, and that they stay synchronized with the code during common refactorings like changing method signature, or moving method to another class. Finally, annotations introduce no performance overhead when the program is executed in non-traced mode.

When a Java program is executed with tracing enabled, the information about the methods participating in tagged program features is recorded in the form of feature traces. A program's functionality can be executed by a test suite, or by a human user. Unlike existing approaches, our approach imposes no requirements for precisely defined execution scenarios, or presence of extensive test suites.

Declaring feature entry points and executing the annotated program are the only manual activities in our remodularization method.

2.2 Feature Representation

The feature representation step of our remodularization method aims at explicitly representing program functionality in the code using *feature modules*. We use Java packages as the mechanism for creating feature modules. The package in the Java programming language is a construct for declaring group of classes, separating the groups from each other in terms of namespace, and controlling access to members of the groups. Our feature representation procedure creates one feature module per program feature, and moves all the participating classes into it. Hence, grouping of classes into feature modules is done accordingly to their participation in the collected feature traces.

One of the issues, which we need to cope with when moving classes into new packages, is the fact that feature traces allow classes and methods to take part in the implementations of multiple features. To assign classes to appropriate feature modules despite of this fact, we use a number of feature membership indicators that we describe in detail in the next section.

The key assumption behind our feature representation procedure is that it is possible to automatically create explicit feature modules, while at the same time preserving the human-understandable concepts existing in the original code. In order to achieve this, we do not split existing classes among the new feature modules. Our motivation is that splitting existing classes would inevitably invalidate their representation of the domain concepts. It needs to be pointed out though that this rather coarse granularity limits the potential of our method for separating features tangled at class, method, or statement level. Since our approach has to assign a single class to a single feature module, some methods will end up placed in incorrect feature modules.

To make the programmers aware of such cases, we mark the methods that are misplaced, or shared between multiple features using two Java annotations: *@Misplaced*, *@Shared*. We parameterize these annotations with identifiers of features in whose implementations the target methods participate. The explicit indication of these kinds of methods allows a programmer to discover and reason about such inter-feature relations more easily.

Although our representation procedure does not split classes that belong to multiple feature modules, it provides some support to programmers who wish to split classes manually. We generate a list of classes that are potential subjects to refactoring. This list is sorted according to the estimated splitting effort. The effort is proportional to the number of features that are tangled in methods of a subject class. We determine the concrete values using a measure for feature tangling in terms of class methods $ftang_{method}$. We base the formulation of it on the general *feature tangling indicator* defined in [15]. After a programmer chooses the candidate class to split, he is guided in determining the division of the class between features by the mentioned indications of misplaced and shared methods.

3. IMPLEMENTATION

In this section we describe the implementation details of the two subsequent steps of our remodularization method: feature location and feature representation.

3.1 FeatureTracer

We have implemented *FeatureTracer* - a feature location library based on the proposed notion of feature entry points. A program that is subject to feature location should be annotated with *FeatureEntryPoint* annotations prior to using the library. Based on that, FeatureTracer detects the execution of features at runtime. The code responsible for execution tracing is implemented as an *aspect* using *AspectJ*, an aspect-oriented programming tool for Java [16].

A Java program is executed in the traced mode by using AspectJ load-time weaver to instrument program classes with a tracing aspect. The aspect matches every call and execution in the Java program. The tracing code extracts declarations of feature entry points from program methods and constructors, and identifies caller-callee pairs of methods, objects and classes. The information about the features being executed at a given time is maintained on thread basis. The collected trace data is used to build a set of *feature trace models* – one model instance per feature. The definition of the model is given in Figure 1.

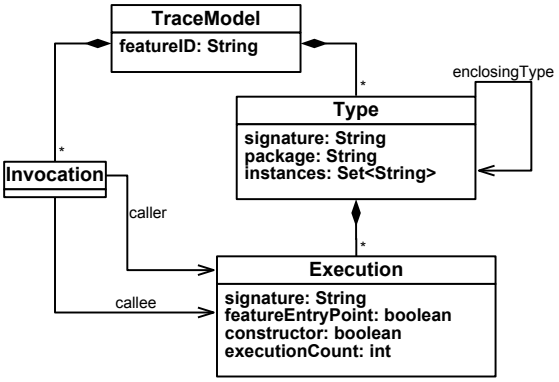


Figure 1. Feature trace model

Feature trace models capture the executions of methods and constructors that occurred at run-time in context of their corresponding features. A method is placed in a feature trace only if it was executed at least once during program execution – an analogous rule applies to types. In case of feature reentries or multithreaded executions, the new trace data is aggregated so that a one-to-one correspondence between model instances and features is preserved. Collected feature-trace models are automatically saved to files during program termination.

In the special cases of abstract methods, inherited methods, and polymorphic method invocations, we follow the rule that feature traces should only capture the methods and types that are executed. This means that for every executed method we register the signature of a class that defines that method's body, and not only inherits or declares it. However, this implies that interfaces will never be registered in feature traces, even though they play an important role as abstractions for concrete classes. We compensate for this in the feature-representation step.

Since one possible use of FeatureTracer is to trace a Java program while a user is interacting with it, the robustness in terms of memory usage over time becomes a serious concern. We avoid the proportionality relation between FeatureTracer's memory usage and the duration of subject program execution by neither collecting information about the order of method invocations, nor timing information. The library increases its memory footprint only for objects and methods that are unique to traces.

3.2 Feature Representation

In order to establish a representation of features in program code we create a new package structure based on the collected feature traces. All source code modifications are done using Spoon program processor library [17].

For every feature identifier found in feature traces, we create a corresponding feature module in terms of a Java package named after the identifier. Program types are then assigned to feature modules based on the *category* they belong to. For the purpose of type categorization we adopt the four-category scheme defined in [18], and supplement it with three new categories of types: *single-feature entry point*, *single-feature instantiated*, and *feature-referenced*. The handling of the types belonging to the distinct categories is summarized in Table 1.

Table 1. Assignment of type categories

Type category	Destination package
Single-feature	Corresponding feature module
Single-feature entry point	
Single-feature instantiated	
Group-feature	Based on dynamic dependencies
Infrastructural	*Infrastructure* module
Feature-referenced	Based on static dependencies
Non-participating	Type's original module

Single-feature types are the types present in execution traces of exactly one feature [18]. This means they are not shared among implementations of multiple features, and therefore can be assigned to their respective feature modules.

Single-feature entry point types are the types that contain feature entry points of only one feature. The methods marked as feature entry points are assumed to have particular importance in implementation of features, since they are explicitly chosen by programmers during the feature location phase.

Instantiation only in a single feature gives a strong indication of membership in that feature's implementation. Furthermore, if such a type does not contain any feature entry points, then it is assumed to belong to implementation of the feature instantiating it.

Group-feature types are present in execution traces of more than one but less that 50% of features [18]. Each type from this category that does not have any of the previously mentioned feature membership indicators is assigned to one of the feature modules based on the number of dynamic dependencies incoming from the feature modules. The dynamic dependencies consist of inter-method invocations collected in the feature traces.

Infrastructural types participate in implementations of 50%, or more, of program features [18]. We assign these types to one common "*infrastructure*" module, as they do not belong to any particular program feature.

By **feature-referenced types** we understand the types that are not present in the feature traces, but are statically referenced by the methods, or type declarations contained in the feature traces. Taking these types into account allows us to reduce the impact of particular realization of program execution on identified implementations of features. A good example of the types that are often not present in feature traces, but should be treated as belonging to implementations of features are Java *exception* types. During program execution, they are referenced dynamically only when they should be thrown, which is likely to occur seldom. All the types that do not get referenced due to a particular realization of the execution flow inside methods are assigned to existing feature modules that statically reference them the most. Interface types, which are not registered in feature traces, also belong to this category.

Non-participating types are the types that are not present in the collected feature traces, and are not feature-referenced types. Non-participating types are an outcome of the fact that we cannot assume full coverage of program code with usage of program execution tracing. All types that are not covered by feature traces are left in their original packages, since there is no indication of their membership in any of the features.

3.2.1 Access Control at Package Boundaries

As our remodularization method alters the existing package structure of the program subject to remodularization, we need to use the access control mechanisms available in Java in a proper way.

First of all, we deal with the issue of the *default* and *protected* scope modifiers. If they are present in the program, they put constraints on the remodularization process, since not every assignment of types to packages can be compiled. We handle the default and protected scope declarations by increasing the visibility of their corresponding Java code elements to *public*. In our opinion it is fully justified to do so, because by changing the criteria for program modularization, one also changes the criteria for access control at boundaries of modules.

Furthermore, the scope declarations can be used to improve encapsulation of feature implementations in the resulting decomposition. We do that by reducing the visibility of types and methods that are used exclusively by single features from public to package-scoped.

4. CASE STUDY

In the presented case studies we evaluate our remodularization approach in terms of required manual effort and quality of the established decompositions. The case studies were conducted on two real-world programs: BlueJ and JHotDraw.

The first case study is concerned with BlueJ [19], an open-source interactive programming environment created to help learning the basics of object-oriented programming in Java. Since BlueJ is a nontrivial application, whose operation involves compilation, dynamic loading, execution, and debugging of Java code, we were particularly eager to test our FeatureTracer library on it. In our case studywe have used BlueJ version 2.5.2.

The second case study investigates JHotDraw [20] - an open-source graphical framework created as an example of a well-designed object-oriented application. Its wide usage in software case studies and the claimed property of good design was what motivated us to include it in our case study. We focus our investigations on the example application *SVG*, which is based on the framework and distributed together with it. The version used here is 7.2.

4.1 Manual Effort

During the manual part of the remodularization process, we needed to identify features, annotate feature entry points, and trigger features from the GUI for both programs.

In the case of BlueJ we were able to infer almost all program features from the available user documentation, since it is written in form of usage scenarios. For JHotDraw such user documentation did not exist, and therefore we needed to rely on the contents of program menus, contextual menus, and toolbars. We followed a procedure of identifying all possible use-cases, and then grouping them into semantically coherent sets constituting features. For instance, two low-level use-cases like "*create project*", "*remove project*" would be grouped as a single "*project management*" feature.

To make the process of annotating feature entry points as consistent as possible, we have forged two rules. Firstly, for each use-case in each feature we tried to find two feature entry points: one in the GUI classes, which most of the time would be the "*actionPerformed*" method of a corresponding *action listener*, and another feature entry point that would actually perform the requested action on a program's domain model. The second rule was to not restrict ourselves to only two annotations per use-case. For instance, if we discovered multiple entry points to the same usage scenario, we annotated all of them. These two rules ensure that even if the program control flow enters a feature's implementation in another way than through its action listeners, the execution of the feature will still be recognized.

We have collected feature traces of the two annotated programs by manually executing all the identified use-cases using the programs' GUIs. No significant performance overhead was observed due to usage of tracing.

Table 2. Summary of the manual effort

	BlueJ	**JHotDraw**
Program size	78 KLOC	62 KLOC
Number of features	38	28
Number of use-cases	121	80
Number of feature entry points	228	91
Total time of indentifying features and use-cases	2 hours	1 hour
Total time of annotating feature entry points	6 hours	4 hours

A summary of the performed manual work is shown in Table 2. The table depicts the effort required to enumerate the programs' functionality, and to place the feature entry point annotations on appropriate methods. It is worth mentioning that we did not know the code, or the architectures of the two programs beforehand.

4.2 Results

By executing our remodularization method on BlueJ and JHotDraw, we obtained the results presented in this section.

In Table 3 we show the distribution of top-level types in the programs among the categories described in Section 3.2. In addition we include the count of types that were found to be "*dead code*" – meaning they are not referenced from the main codebases

of the programs. For BlueJ the majority of these types are in package *org.syntax.jedit.tokenmarker*, whereas for JHotDraw these are the framework classes not used by the investigated *SVG* application.

Table 3. Number of top-level types in type categories

Type category	BlueJ	JHotDraw
Single-feature	45	57
Single-feature entry point	35	20
Single-feature instantiated	27	19
Group-feature	159	100
Infrastructural	9	12
Feature-referenced	129	88
Non-participating	97	37
"Dead code"	34	161

The main results of our case study are shown in Table 4. The table presents the quantifiable impact of our remodularization method on the quality of feature representation in subject programs. For this purpose we have used the notions of feature *tangling* and *scattering* in terms of packages $fsca_{pkg}$, $ftang_{pkg}$ as defined in [15]. Since we aim to improve representation of features, we want to reduce scattering and tangling of feature implementations.

Table 4. Summary impact on feature representation's quality

	BlueJ			JHotDraw		
	pre	**post**	**change**	**pre**	**post**	**change**
$fsca_{pkg}$	0.230	0.168	-27%	0.280	0.160	-43%
$ftang_{pkg}$	0.300	0.245	-18%	0.367	0.281	-23%

A more detailed overview of how scattering and tangling changes during the remodularization process is given in Figures 2-5. These figures visualize the distributions of scattering and tangling in both subject programs before and after remodularization. Scattering of feature implementations is presented on feature-basis, allowing for direct comparison between the original and the remodularized versions of the programs. For tangling, we give only a general overview of the distribution shapes, as no traceability can be established between package structures of the programs pre and post the remodularization process.

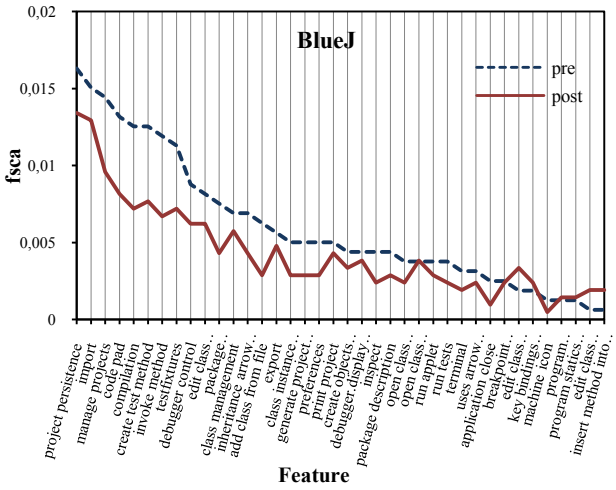


Figure 2. Scattering of features in BlueJ

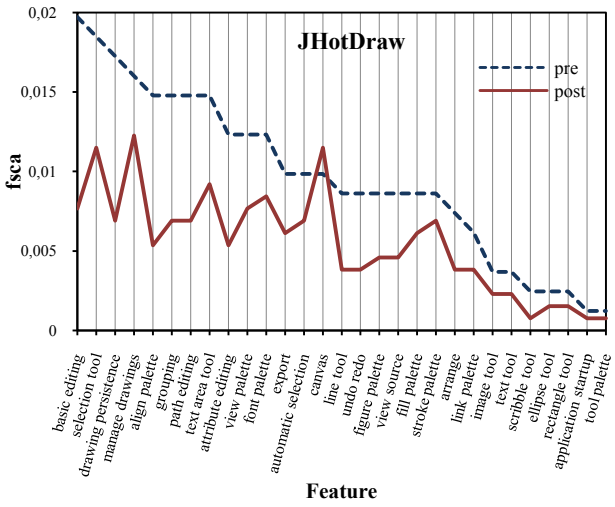


Figure 3. Scattering of features in JHotDraw

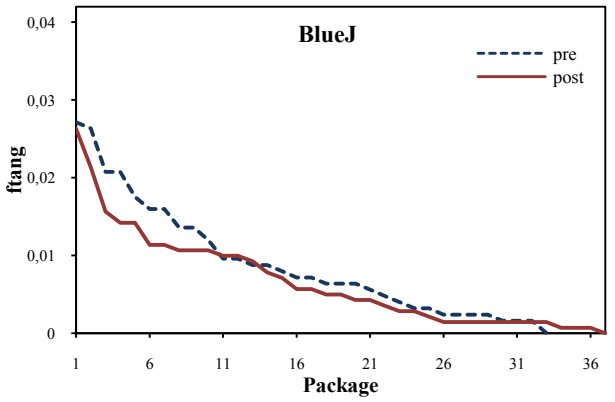


Figure 4. Tangling of features in packages of BlueJ

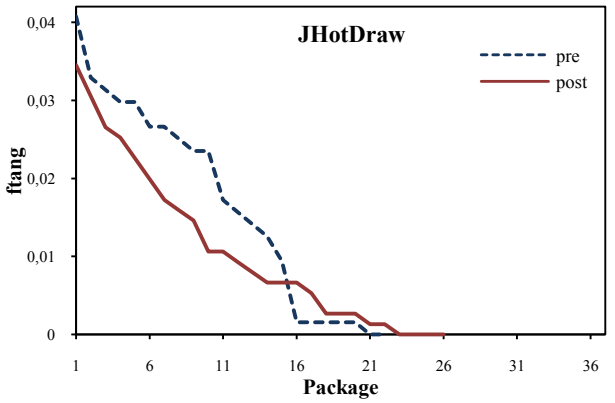


Figure 5. Tangling of features in packages of JHotDraw

4.3 Discussion

Comparison between the original programs and their remodularized versions reveals that our approach has reduced the total tangling and scattering of feature implementations. For BlueJ, scattering of feature implementations was decreased by 27%, whereas in case of JHotDraw it was reduced by 43%. Tangling of feature implementations in terms of program packages was diminished by 18% for BlueJ and by 23% for JHotDraw. These results indicate that our remodularization approach has significantly improved representation of features in the two programs, despite of avoiding the splitting of classes and methods.

Presented plots of feature implementations' scattering over programs' packages indicate that our method has reduced the scattering for the majority of features in both programs. However, there exist a total of four features for which scattering was increased. In case of BlueJ the three negatively impacted features handle editing of class implementations in BlueJ's source code editor, whereas in case of JHotDraw the feature that became more scattered is the "*canvas*" feature responsible for displaying and changing properties of the drawing canvas. At present we do not have any confirmed explanation of this phenomenon, but we plan to investigate it further.

Last but not least, we believe that the manual effort needed to annotate both programs with feature entry points turns out to be relatively low. In total we needed 13 hours to finalize this task. We have found the required manual work to be simple, and not requiring extensive knowledge of the subject code. Most importantly, we have found FeatureTracer to be non-invasive in a sense that it did not oblige us to modify any of the original method implementations. Thus we are confident that the correctness of the subject programs was not altered during the process.

One issue that we have experienced with the proposed feature-location procedure is the inability to consistently repeat the feature-annotation process. This was found to impede reproducibility of the remodularization results. However, no formal investigation of this issue was conducted in the context of this case study.

5. RELATED WORK

An approach, which is strongly related to ours, was described in [21]. The authors present a method for locating and refactoring features into fine-grained components. Their approach emphasizes improvement of evolvability as the primary reason for remodularization, which is closely related to our point of view. However, the two approaches have different purposes, and different levels of automation. The approach presented in [21] aims at extracting program features and refactoring them into reusable components. Feature location is done by triggering runtime traces using a dedicated regression test suite. Feature implementations are then manually analyzed, and manually refactored into components, according to a proposed component model.

Feature location based on feature entry points overcomes some of the limitations of the existing approaches. FeatureTracer does not rely on extensive test suites as in [22], manual marking of complete feature implementations in IDE [23], or external meta-data descriptors [24][25]. Apart from the small manual workload required, our location procedure has two more advantages over the existing approaches. The mapping of feature entry points to program methods will be preserved during program refactoring, which is not necessarily the case for approaches like the mentioned IDE-marking, or the external meta-data descriptors. Last but not least, user-driven usage of FeatureTracer allows for tracing GUI classes, which can be difficult to achieve with test suite-driven triggering (e.g. [22]).

A large portion of the existing feature representation research has been carried out in the area of *feature-oriented refactoring* (*FOR*) [23]. Feature-oriented refactoring is the process of decomposing a program into features, where a feature is an increment in program functionality [23]. This definition implies that features, as functionality increments, do not share any code with each other. This is opposed to the definition used in this work, where we acknowledge that a fragment of code can participate in the implementation of multiple features. Based on the mentioned formulation, the FOR approach aims at a complete separation of features' implementations. Realizations of this idea were shown to be successfully created using diverse tools: *AHEAD* [23][26], AspectJ [9], *Hyper/J* [9], *Jiazzi* [25], and *Caesar* [27].

Our approach differs from FOR in one significant way. In our method we aim at preserving existing concepts expressed in the code through types, where the FOR aims at achieving a complete separation of feature implementations through modification of existing concepts. In case of FOR, preservation of human-understandable abstractions is usually not a requirement, since the goal is to compose product line members in an automated fashion, as shown for instance in [23]. In contrast, our method is targeted at supporting traditional development and maintenance, during which program comprehension is an important factor. However, we achieve that at a cost of not being able to fully separate feature implementations.

6. CONCLUSION AND FUTURE WORK

As mentioned previously, current techniques for remodularization of object-oriented software towards feature decomposition require a substantial amount of manual effort, and invalidate concepts existing in the original code.

In this paper we have proposed a semi-automatic remodularization method that does not have these limitations. The method involves non-invasive, low-workload feature location, and explicit feature representation based on Java packages. Our method does not split existing classes in order to preserve human-understandable concepts that are present in the original decompositions of subject programs in form of their domain models.

We have demonstrated that the proposed method successfully improves feature representation in a static program structure in terms of reducing feature implementations' scattering and tangling.

Our approach makes it possible to switch from standard program decomposition to feature decomposition without imposing an extensive workload on the programmer. Thereby, we believe that our method is well suited for supporting the activities of change adoption and error correction during software maintenance.

Our future plans include attempts to further reduce the manual effort required by our remodularization approach. We would like to experiment with the usage of the *canonical features* [28] as a basis for creating feature modules. Another direction is to enumerate ideas for automated placement of feature entry points

in program code, which can help to improve reproducibility of the process. Finally, we plan to seek for a lightweight, Java-based approach for splitting feature implementations, which would allow for preservation of existing code concepts.

7. AVAILABILITY

The FeatureTracer library can be obtained by contacting the authors. We also plan to release it as an open-source project in the near future.

8. REFERENCES

[1] Bennett, K. H. and Rajlich, V. T. 2000. Software maintenance and evolution: a roadmap. In Proceedings of the Conference on the Future of Software Engineering (Limerick, Ireland, June 04 - 11, 2000). ICSE '00. ACM, New York, NY, 73-87.

[2] Turner, C. R., Fuggetta, A., Lavazza, L., and Wolf, A. L. 1999. A conceptual basis for feature engineering. *J. Syst. Softw.* 49, 1 (Dec. 1999), 3-15.

[3] Shaft, T., and Vessey, I. 2006. The role of cognitive fit in the relationship between software comprehension and modification. *MIS Quarterly*, 30, 1 (2006), 29-55.

[4] Krasner, G. E. and Pope, S. T. 1988. A cookbook for using the model-view controller user interface paradigm in Smalltalk-80. *J. Object Oriented Program.* 1, 3 (Aug. 1988), 26-49.

[5] Röthlisberger, D., Greevy, O., and Nierstrasz, O. 2007. Feature driven browsing. In *Proceedings of the 2007 international Conference on Dynamic Languages: in Conjunction with the 15th international Smalltalk Joint Conference 2007* (Lugano, Switzerland, August 25 - 31, 2007). ICDL '07, vol. 286. ACM, New York, NY, 79-100.

[6] Cornelissen, B., Zaidman, A., Van Rompaey, B., and van Deursen, A. 2009. Trace visualization for program comprehension: A controlled experiment. In *Proceedings of the 17th International Conference on Program Comprehension*. ICPC. IEEE Computer Society, 2009, pp. 100-109.

[7] Parnas, D. L. 1972. On the criteria to be used in decomposing systems into modules. *Commun. ACM* 15, 12 (Dec. 1972), 1053-1058.

[8] Greevy, O., Girba, T., and Ducasse, S. 2007. How Developers Develop Features. In *Proceedings of the 11th European Conference on Software Maintenance and Reengineering* (March 21 - 23, 2007). CSMR. IEEE Computer Society, Washington, DC, 265-274.

[9] Murphy, G. C., Lai, A., Walker, R. J., and Robillard, M. P. 2001. Separating features in source code: an exploratory study. In *Proceedings of the 23rd international Conference on Software Engineering* (Toronto, Ontario, Canada, May 12 - 19, 2001). ICSE. IEEE Computer Society, Washington, DC, 275-284.

[10] Ossher, H., and Tarr, P. 2000. On the need for on-demand remodularization. In *ECOOP'2000 workshop on Aspects and Separation of Concerns* (2000).

[11] Korson, T. and McGregor, J. D. 1990. Understanding object-oriented: a unifying paradigm. *Commun. ACM* 33, 9 (Sep. 1990), 40-60.

[12] Rajlich, V. and Wilde, N. 2002. The Role of Concepts in Program Comprehension. In *Proceedings of the 10th international Workshop on Program Comprehension* (June 27 - 29, 2002). IWPC. IEEE Computer Society, Washington, DC, 271.

[13] Wilde, N., Gomez, J. A., Gust, T., and Strasburg, D. 1992. Locating user functionality in old code. In *Proceedings of International Conference on Software Maintenance* (1992), 200-205.

[14] Biggerstaff, T. J., Mitbander, B. G., and Webster, D. 1993. The concept assignment problem in program understanding. In *Proceedings of the 15th international Conference on Software Engineering* (Baltimore, Maryland, United States, May 17 - 21, 1993). International Conference on Software Engineering. IEEE Computer Society Press, Los Alamitos, CA, 482-498.

[15] Brcina, R., Riebisch, M. 2008. Architecting for evolvability by means of traceability and features. In *23rd IEEE/ACM International Conference on Automated Software Engineering - ASE Workshops* (2008), 72-81.

[16] Kiczales, G., Irwin, J., Lamping, J., Loingtier, J., M., Lopes, C., V., Maeda, C., and Mendhekar, A. 1996. Aspect-oriented programming. *ACM Computing Surveys* (1996), vol. 28.

[17] http://spoon.gforge.inria.fr/

[18] Greevy, O. and Ducasse, S. 2005. Correlating Features and Code Using a Compact Two-Sided Trace Analysis Approach. In *Proceedings of the Ninth European Conference on Software Maintenance and Reengineering* (March 21 - 23, 2005). CSMR. IEEE Computer Society, Washington, DC, 314-323.

[19] http://www.bluej.org/

[20] http://www.jhotdraw.org/

[21] Mehta, A. and Heineman, G. T. 2002. Evolving legacy system features into fine-grained components. In *Proceedings of the 24th international Conference on Software Engineering* (Orlando, Florida, May 19 - 25, 2002). ICSE '02. ACM, New York, NY, 417-427.

[22] Wilde, N. and Scully, M. C. 1995. Software reconnaissance: mapping program features to code. *Journal of Software Maintenance* 7, 1 (Jan. 1995), 49-62.

[23] Liu, J., Batory, D., and Lengauer, C. 2006. Feature oriented refactoring of legacy applications. In *Proceedings of the 28th international Conference on Software Engineering* (Shanghai, China, May 20 - 28, 2006). ICSE '06. ACM, New York, NY, 112-121.

[24] Tarr, P., Ossher, H., Harrison, W., and Sutton, S. M. 1999. N degrees of separation: multi-dimensional separation of concerns. In *Proceedings of the 21st international Conference on Software Engineering* (Los Angeles, California, United States, May 16 - 22, 1999).

[25] McDirmid, S., Flatt, M., and Hsieh, W. C. 2001. Jiazzi: new-age components for old-fasioned Java. In *Proceedings of the 16th ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications* (Tampa Bay, FL, USA, October 14 - 18, 2001). OOPSLA '01. ACM, New York, NY, 211-222.

[26] Trujillo Trujillo, S., Batory, D., and Diaz, O. 2006. Feature refactoring a multi-representation program into a product line. In *Proceedings of the 5th international Conference on Generative Programming and Component Engineering* (Portland, Oregon, USA, October 22 - 26, 2006). GPCE '06. ACM, New York, NY, 191-200.

[27] Mezini, M. and Ostermann, K. 2004. Variability management with feature-oriented programming and aspects. In *Proceedings of the 12th ACM SIGSOFT Twelfth international Symposium on Foundations of Software Engineering* (Newport Beach, CA, USA, October 31 - November 06, 2004). SIGSOFT '04/FSE-12. ACM, New York, NY, 127-136.

[28] Kothari, J., Denton, T., Shokoufandeh, A., and Mancoridis, S. 2007. Reducing Program Comprehension Effort in Evolving Software by Recognizing Feature Implementation Convergence. In *Proceedings of the 15th IEEE international Conference on Program Comprehension* (June 26 - 29, 2007). ICPC. IEEE Computer Society, Washington, DC, 17-26.

An Orthogonal Access Modifier Model for Feature-Oriented Programming

Sven Apel and Jörg Liebig
Department of Informatics and Mathematics
University of Passau, Germany
{apel, joliebig}@fim.uni-passau.de

Christian Kästner and Martin Kuhlemann
School of Computer Science
University of Magdeburg, Germany
{kaestner, kuhlemann}@iti.cs.uni-magdeburg.de

Thomas Leich
Metop Research Center
Magdeburg, Germany
thomas.leich@metop.de

ABSTRACT

In *feature-oriented programming* (*FOP*), a programmer decomposes a program in terms of features. Ideally, features are implemented modularly so that they can be developed in isolation. Access control is an important ingredient to attain feature modularity as it provides mechanisms to hide and expose internal details of a module's implementation. But developers of contemporary feature-oriented languages did not consider access control mechanisms so far. The absence of a well-defined access control model for FOP breaks the encapsulation of feature code and leads to unexpected and undefined program behaviors as well as inadvertent type errors, as we will demonstrate. The reason for these problems is that common object-oriented modifiers, typically provided by the base language, are not expressive enough for FOP and interact in subtle ways with feature-oriented language mechanisms. We raise awareness of this problem, propose three feature-oriented modifiers for access control, and present an orthogonal access modifier model.

Categories and Subject Descriptors: D.2.2 [**Software**]: Software Engineering—*Design Tools and Techniques*; D.3.3 [**Software**]: Programming Languages—*Language Constructs and Features*

General Terms: Design, Languages

Keywords: Feature-Oriented Programming, Orthogonal Access Modifier Model

1. INTRODUCTION

The goal of *feature-oriented programming* (*FOP*) is to modularize software systems in terms of features [19, 11]. A *feature* is a unit of functionality of a program that satisfies a requirement, represents a design decision, and provides a potential configuration option [2]. A *feature module* encapsulates exactly the code that contributes to the implementation of a feature [8]. The goal of the decomposition into feature modules is to construct well-structured software that can be tailored to the needs of the user and the application scenario. Typically, from a set of feature modules, many different programs can be generated that share common features and differ in other features, which is also called a software product line [12, 18].

Many feature-oriented languages aim at feature modularity, e.g, AHEAD/Jak [11], FeatureC++ [7], and FeatureHouse [6]. Feature modules are supposed to hide implementation details and to provide access via interfaces. The rationale behind such information hiding is to allow programmers to develop, type check, and compile features in isolation. However, contemporary feature-oriented languages do not perform well with regard to feature modularity [16]; they lack sufficient abstraction and modularization mechanisms to support (1) independent development based on information hiding, (2) modular type checking, and (3) separate compilation. In a theoretical work, Hutchins has shown that, in principle, feature-oriented languages should be able to attain this level of feature modularity [14]. However, there are many open issues regarding the implementation on the basis of a mainstream programming language, such as the interaction with other language mechanisms, efficiency, and tool support.

An important ingredient for feature modularity that is missing in contemporary feature-oriented languages is a proper mechanism for access control. Access modifiers allow programmers to define the scope and visibility of their program elements such that implementation details can be encapsulated. For example, in Java, programmers use access modifiers (e.g., `private` or `public`) to grant or prohibit access to classes, methods, and fields. However, there are no specific modifiers tailored to feature-oriented language mechanisms. Well, since a feature-oriented language usually extends an object-oriented language (e.g., Jak extends Java [11] and FeatureC++ extends C++ [7]), the object-oriented access modifiers are (re)used. But it is not possible to grant access, e.g., to a program element for all other program elements from the same feature and to disallow the access for all program elements of other features.

FOSD'09, October 6, 2009, Denver, Colorado, USA.

As said before, access control has not been considered so far in research on feature-oriented languages. In some sense access control mechanisms were for free when extending an existing object-oriented language. Of course, the object-oriented modifiers were not intended for the use in FOP, so one can say that they are misused. We contribute an analysis of object-oriented modifiers used in FOP and identify several shortcomings and problems that lead to a limited expressiveness of feature-oriented languages, unexpected and undefined program behaviors, and inadvertent type errors. We explore the design space of feature-oriented access control mechanisms and propose three concrete access modifiers. Furthermore, we present an orthogonal access modifier model, which integrates common object-oriented modifiers with our novel feature-oriented modifiers.

2. BACKGROUND

Often, a feature-oriented language extends an object-oriented base language by mechanisms for the abstraction and modularization of features.[1] In order to implement the additions and changes a feature makes, feature-oriented languages like Jak introduce a mechanism for class refinement.

In Figure 1, we depict a class `Stack` written in Jak, which is an extension of Java and belongs to the AHEAD tool suite [11]. The class definition is identical to a definition in Java except for the `layer` declaration, which defines the feature to which class `Stack` belongs – in our case feature BASE.

Feature BASE

```
1  layer Base;
2  class Stack {
3    private LinkedList elements = new LinkedList();
4    public void push(Object element) {
5      elements.addFirst(element);
6    }
7    public Object pop() {
8      if(elements.size() > 0) { return elements.removeFirst(); }
9      else { return null; }
10   }
11 }
```

Figure 1: A basic stack implemented in Jak.

In Figure 2, we depict a refinement of class `Stack`, declared by keyword `refines`. The refinement is part of feature UNDO, which allows the clients of the stack to revert the last operation. When feature UNDO is composed with feature BASE, the refinement adds a new method `undo` and two new fields `lastPush` and `lastPop` to class `Stack`. Furthermore, it refines the methods `push` and `pop` (by overriding) in order to store the last item added to or removed from the stack. Keyword `Super` is used to invoke the method that has been refined.[2]

Typically, a feature comprises multiple class declarations and class refinements, which implement the feature in concert. We visualize a feature-oriented program design – like the design of our stack example – using a *collaboration diagram* [20, 23, 21]. In Figure 3, we show a sample feature-oriented design, which decomposes the underlying object-oriented design into features. The design in Figure 3 consists of the four classes $A - D$ (represented by medium-gray boxes), which are located in the two packages P_1 and P_2 (represented by light-gray boxes). The diagram displays features ($F_1 - F_3$) as slices that cut across the core object-oriented design (represented by dark-gray boxes). Hence, a class is decomposed into several fragments, called *roles*, that belong to different features [23]; the set of roles belonging to a feature is called a *collaboration* [21] and is encapsulated by a feature module [8]. For example, class A consists of the roles A_1, A_2, and A_3; feature F_1 is implemented by the roles A_1, B_1, C_1, and D_1. The top most role of a class is also called the *base class* (e.g., A_1) and the other roles are called *class refinements* (e.g., A_2 and A_3) [11]. The solid arrow denotes the refinement relationship between roles and the empty arrow denotes inheritance between full classes.

Feature UNDO

```
12 layer Undo;
13 refines class Stack {
14   private Object lastPush = null;
15   private Object lastPop = null;
16   public void push(Object item) {
17     lastPush = item; lastPop = null;
18     Super.push(item);
19   }
20   public Object pop() {
21     lastPop = Super.pop();
22     lastPush = null; return lastPop;
23   }
24   public void undo() {
25     if(lastPush != null) { Super.pop(); }
26     else if(lastPop != null) { Super.push(lastPop); }
27   }
28 }
```

Figure 2: A refinement of class `Stack` implemented in Jak.

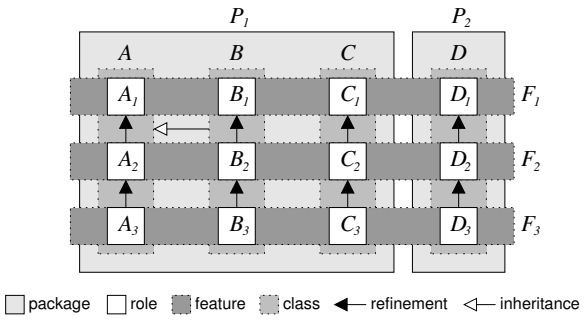


Figure 3: A sample feature-oriented design.

3. PROBLEM STATEMENT

We explain the problems we encountered with feature-oriented languages by means of Jak. Jak, as a Java extension, has inherited the access modifiers of Java. Hence, programmers can control the access to classes and members in Jak using the modifiers `private`, `protected`, `package`, and `public`.[3] But there are two problems with this:

[1] We are aware that some feature-oriented tools build on languages that are not object-oriented [11, 1, 6]. These languages are outside the scope of the paper, as they do not provide access modifiers like the ones we consider here.

[2] Note that, for brevity, we use a slightly less verbose notation than in Jak; other feature-oriented languages use different keywords anyway.

[3] We assume a basic knowledge on Java's access modifiers. In Java, if a class, field, or method does not have an access modifier then only elements from the same package may ac-

1. **Undefined semantics**: object-oriented modifiers interact in undefined ways with feature-oriented mechanisms such as class refinements
2. **Limited expressiveness**: object-oriented modifiers are not expressive enough to control the access to elements introduced by features.

Undefined Semantics

Let us illustrate the first problem by means of our stack example. Suppose we refine our class `Stack` by applying a feature Trace. Feature Trace monitors the accesses to the stack and, as soon as the stack is changed, it writes all stack elements to the console. In Figure 4, we depict a corresponding refinement, which refines the methods `push` and `pop`, accesses the list storing the stack's elements, and prints them to the console.

Feature Trace

```
1  layer Trace;
2  refines class Stack {
3    public void push(Object item) {
4      Super.push(item);
5      trace();
6    }
7    public Object pop() {
8      Object res = Super.pop();
9      trace(); return res;
10   }
11   private void trace() {
12     for(int i = 0; i < elements.size(); i++) {
13       System.out.print(elements.get(i).toString() + " ");
14     }
15   }
16 }
```

Figure 4: A refinement of class `Stack` to trace accesses to a stack instance.

The question is whether the above example is correct. Is it allowed for the class refinement to access the private field `elements` of the refined class? The answer is not obvious since feature-oriented languages usually do not come with a specification (the behavior is de facto defined by the implementation of the composition engine) and formally specified subsets of feature-oriented languages do not include modifiers [5, 13, 4]. Compiling this code (or similar code) with the Jak compiler reveals that it depends on certain compiler flags whether this code is considered correct.

The background is that the Jak compiler generates Java code in an intermediate step and it supports two options to do so [15]: in the first option, called Mixin, the compiler generates an inheritance hierarchy with one subclass per refinement; in the second option, called Jampack, the compiler generates a single class consisting of the elements of the base class and all of its refinements. Comparing the two options it becomes clear why they show different behaviors in our example, which we illustrate in Figure 5. In the first option, private field `elements` cannot be accessed because the refinement is translated to a subclass, which cannot access private members of superclasses. In the second option, private field `elements` can be accessed because all code of all refinements is moved to the class that is refined. So we have two different behaviors of a single program depending on a compiler flag that is intended for optimization.

cess them. For sake of symmetry with the other modifiers, we introduce modifier `package` for this case.

```
1  class StackBase {
2    private LinkedList elements ...
3  }
4  class Stack extends StackBase {
5    ...
6    private void trace() {
7      ... elements.size() ...
8      ... elements.get(i) ...
9    }
10 }
```

```
1  class Stack {
2    private LinkedList elements ...
3    ...
4    private void trace() {
5      ... elements.size() ...
6      ... elements.get(i) ...
7    }
8  }
```

Figure 5: Mixin vs. Jampack.

One can argue for one or the other behavior, and certainly it is possible to fix either Mixin or Jampack such that both obey an equal behavior, but what we would like to stress is that the semantics of access modifiers and their interaction with feature-oriented mechanisms such as class refinements is not well-defined. This fact is not only a matter of tool support since it can affect the program semantics beyond type errors. Have a look at the example shown in Figure 6.[4] Which value is returned by method `bar`? Again, it depends on the composition mechanism: using Jampack, `bar` returns 23; using Mixin, `bar` returns 42. A comprehensive discussion of the reason of difference is outside the scope of the paper and we leave it as "homework" for the reader. A hint is that it depends again on the underlying composition mechanism (Mixin-like or Jampack-like) and that it has to do with Java's overloading mechanism.

Feature Base

```
1  layer Base;
2  class A {}
3  class B extends A {}
4  class Foo {
5    protected int foo(A a) { return 23; }
6    private int foo(B b) { return 42; }
7  }
```

Feature Ext

```
8  layer Ext;
9  refines class Foo {
10   public int bar() { return foo(new B()); }
11 }
```

Figure 6: Which value is returned by method `bar`?

In Table 1, we compare different (variants of) feature-oriented languages with respect to their rules for accessing fields from a refinement and the program behavior with respect to our example of Figure 6. We argue that the differences between the individual (variants of) feature-oriented languages are not intended but stem solely from the fact that research on FOP did not consider access modifiers so far. The language developers got modifiers for free from the base language and the implementation of the composition in a preprocessing step decides over the semantics of the composed program.

We hope that the above examples make clear that we need well-defined semantics of feature-oriented languages including access modifiers as well as a scientific discussion that motivates the choices of the semantics definition. What we do not want is that internal implementation details of compilers or the use of compiler flags, which target at optimization [15], decide arbitrarily over the program semantics.

[4] For brevity we have merged the definitions of the classes `A`, `B`, and `Foo` in a single listing.

	Jak[1] (Mixin)	Jak[1] (Jampack)	FeatureHouse[2]	FeatureC++[3]	Classbox/J[4]	Caesar J[5]	OT/J[6]
`private`	×	✓	✓	✓	✓	×	✓
`protected`	✓	✓	✓	✓	✓	✓	✓
`package`	✓	✓	✓	—	✓	—	✓
`public`	✓	✓	✓	✓	✓	✓	✓
`bar()` (Fig. 6)	23	42	42	42	42	23	42

[1] `http://www.cs.utexas.edu/~schwartz/ATS.html`
[2] `http://www.fosd.de/fh/`
[3] `http://wwwiti.cs.uni-magdeburg.de/iti_db/forschung/fop/featurec/`
[4] `http://scg.unibe.ch/research/classboxes/`
[5] `http://caesarj.org/`
[6] `http://www.objectteams.org/`

Table 1: Which members of a class can be accessed by a refinement? What is the return value of `bar`? (× access prohibited; ✓ access granted; — not supported)

Limited Expressiveness

With regard to the second problem (object-oriented modifiers are not expressive enough for feature-oriented mechanisms), consider the following example. Suppose we refine our class `Stack` such that accessing the stack's methods is thread-safe. The refinement shown in Figure 7 adds a new field `lock` and overrides the methods `push` and `pop` in order to synchronize access via the methods `lock` and `unlock`. Furthermore, suppose that feature Sync also refines many other classes in order to attain thread safety (e.g., `Queue`, `Map`, and `Set`) and that a central registry keeps track of all locks in use. In order to grant the lock registry access to the lock fields of the synchronized stack (queue, map, set, ...) objects, we have to change the access modifier in Line 3 from `private` to `public` (similarly for the other synchronized classes). However, this also means that every class of the entire program has access to the lock (not only the lock registry), which is certainly not desired. Other modifiers such as `package` and `protected` are not sufficient as well, which is easy to see and omitted for brevity. Instead, we envision a modifier that states that all roles of a given feature may access a member within the same feature. In our case, we would like to grant access to the locks from the lock registry, which is introduced in the same feature as the locks are. The synchronization example illustrates that the access modifiers available in contemporary feature-oriented languages are not sufficient for fine-grained, feature-based access control.

Feature Sync

```
1  layer Sync;
2  refines class Stack {
3    private Lock lock = new Lock();
4    public void push(Object item) {
5      lock.lock();
6      Super.push(item);
7      lock.unlock();
8    }
9    public Object pop() {
10     lock.lock();
11     Object res = Super.pop();
12     lock.unlock(); return res;
13   }
14 }
```

Figure 7: A refinement of class `Stack` to synchronize accesses to a stack instance.

Summary

Our previous discussion shows that we need access modifiers that are specific to the needs of FOP. Programmers would like to provide access to a program element from certain features. Furthermore, we would like to define how the feature-oriented modifiers interplay with the object-oriented modifiers in order to avoid inadvertent interactions. To this end, in the next section, we define an orthogonal access modifier model for feature-oriented languages.

4. AN ORTHOGONAL ACCESS MODIFIER MODEL

Next, we explore the design space of possible and potentially useful modifiers for feature-oriented language mechanisms. First, we introduce three feature-oriented modifiers and, second, we explain how they can be combined with the modifiers commonly found in object-oriented languages.

4.1 Feature-Oriented Modifiers

Using the sample feature-oriented design of Figure 3, we explain three possible modifiers that control the access to members of roles. The motivation for the modifiers comes directly from the fact that features cut across the underlying object-oriented design.

Modifier `feature`

The idea for modifier `feature` is motivated by our example, in which we added synchronization support to a stack and other data structures. There we had the problem that with object-oriented modifiers we were not able to express that only elements introduced by the synchronization feature may access the lock fields of the refined classes. The modifier `feature` grants exactly this access and forbids the access from other features, as we illustrate for our stack example in Figure 8. Modifying a member with `feature` allows every other role of the same feature to access the member in question, in our example, including the lock registry.

Modifier `subsequent`

The proposal of modifier `subsequent` is motivated by the fact that some FOP approaches treat features as stepwise refinements. That is, starting from a base program, features gradually refine the existing program code and pro-

Feature Sync

```
1 layer Sync;
2 refines class Stack {
3   feature Lock lock = new Lock();
4   ...
5 }
```

Figure 8: Using modifier `feature` to grant access to field `lock` from all members of feature Sync.

duce in each step a new version [11, 17]. Some researchers even draw a connection to functions that map programs to programs [10, 11, 17]. In the stepwise refinement scenario, it has been argued that a feature (represented by a function) does never "know" about program elements applied by feature that have been applied subsequently. The positive effect of such a disciplined programming style is that inadvertent interactions cannot occur with program elements that are not known at the development time of a feature [17]. This is especially important for languages that support a pattern-based selection of extension points such advice and implicit invocation [22,3], which have been discussed recently in the context of FOP [17,8]. In order to support this view, we propose a modifier `subsequent` that grants access to a program element from all elements of the same feature or of features added subsequently. Features that have been added previously cannot access the program element in question.

Modifier `program`

Modifier `program` broadens the scope of access to a member from program elements of all features. This is like the current situation in feature-oriented languages where programmers have no fine-grained access control with regard to feature-related code, except that in our novel proposal the semantics of object-oriented modifiers and their interplay with feature-oriented mechanisms is well-defined, which we explain in Section 4.2.

Discussion

A question that arises is whether the new modifiers are expressive enough or whether we need even a more fine-grained access control mechanism. The smallest modularization unit in feature-oriented designs is the role. With our three feature-oriented modifiers, we are able to precisely control the access of individual roles to the elements of another role. So there is no need for a more fine-grained access. At the other end of the spectrum, it is possible to grant universal access, which is like leaving out feature-oriented access modifiers at all. The modifier `subsequent` is in the middle and motivated by previous work on program design. One can imagine a further modifier `previous`, which would be the inverse of `subsequent`, but we argue that such a modifier is not of practical value. Although it has been observed that there are situations, in which a feature access elements that have been introduced later, this is not the rule [3]. In these situations, a programmer can use modifier `program` because it is certainly not meaningful full to forbid the access from subsequent features.

A further possibility would be to grant access only to a special feature or a subset of features. We did not consider this possibility *so far* because we would like to minimize the coupling between feature implementation and feature management. Apart from the `layer` declaration at the beginning of each Jak file, there is no information about the actual features. Instead, the relation between features and code is implicit and managed externally by the tool infrastructure. We believe that this separation of concerns (feature implementation vs. feature management) is one of the success factors for contemporary feature-oriented languages and tools [2]. But the last word is not spoken on this issue.

A_2	`feature`	`subsequent`	`program`
`private`	A_2	A_2, A_3	A_1, A_2, A_3
`protected`	A_2, B_2	A_2, A_3, B_2, B_3	A_1, A_2, A_3, B_1, B_2, B_3
`package`	A_2, B_2, C_2	A_2, A_3, B_2, B_3, C_2, C_3	A_1, A_2, A_3, B_1, B_2, B_3, C_1, C_2, C_3
`public`	A_2, B_2, C_2, D_2	A_2, A_3, B_2, B_3, C_2, C_3, D_2, D_3	A_1, A_2, A_3, B_1, B_2, B_3, C_1, C_2, C_3, D_1, D_2, D_3

Table 2: Overview of the roles that may access a member that has been introduced in role A_2.

Some feature-oriented languages support to modify the access to individual roles, e.g., `public refines class A { ... }`. Using such a modifier in such a position we can subsequently broaden the access to a class. That is, we can make a private class protected or public but not vice versa. Thus, a modifier in such a position does not control the access to program elements of feature-related code, but it overrides an existing object-oriented modifier. This mechanism can also be used to broaden the access to the members of a class.

Finally, it remains open how modifiers like `abstract` and `final` fit into the picture and how they can be combined gainfully with feature-oriented modifiers. We shall address this issue in further work.

4.2 Object-Oriented and Feature-Oriented Modifiers in Concert

We have proposed three feature-oriented access modifiers, which interact with object-oriented modifiers in different ways. In Table 2, we depict the interplay between object-oriented and feature-oriented modifiers with respect to the sample feature-oriented design of Figure 3. For each combination of object-oriented and feature-oriented modifiers, the table shows the roles that may access the members of role A_2 in our sample design of Figure 3. That is, each cell of Table 2 contains the roles that are allowed to access role A_2's members, which have the combined modifiers corresponding to the cell's column and row. For example, a member of role A_2 with the modifiers `protected` and `feature` can be accessed by the roles A_2 and B_2 (first column, second row); a member of role A_2 modified with `private` and `program` can be accessed by the roles A_1, A_2, and A_3 (third column, first row).

Looking closer at Table 2, it is interesting to observe that the individual modifier combinations constitute a lattice with '`private feature`' as bottom element and '`public program`' as top element, as illustrated in Figure 9. The lattice can guide the formalization and implementation of a corresponding type system, which is concerned with the question whether the scope of the requested access is smaller or larger than the one of the accessed element. When a

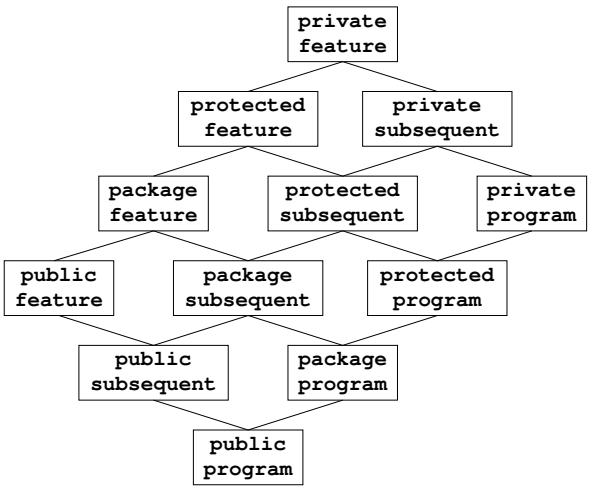


Figure 9: A lattice formed by modifier combinations.

programmer overrides a member, as in the case of method overriding, the scope of the member's access may stay unchanged, can be extended, but cannot be limited, which means the modifier itself or any modifiers below the original modifier.

5. FORMALIZATION AND IMPLEMENTATION ISSUES

Although Table 2 captures the idea of our modifiers nicely, in further work, a formal definition of the operational semantics and type system of a feature-oriented language that supports these modifiers is desirable. This way, we will be able to define the semantics of our modifiers unambiguously and to guide the implementation of feature-oriented compilers. As a formal system, we will use the Feature Featherweight Java (FFJ) calculus [5], which extends a minimal core of Java with feature-oriented mechanisms. The formalization of the orthogonal access modifier model should be straightforward and we believe that we will be able to prove the soundness of the corresponding type system.

We intend to implement a compiler on the basis of an existing feature-oriented language, preferably Jak or FeatureHouse, which can be used for an empirical evaluation. The problem of current language implementations is that they do not provide a type system that takes the feature-oriented abstractions into account. Merely, feature-oriented code is translated to object-oriented code, and an object-oriented compiler type checks the translated code. Since our feature-oriented modifiers do not have corresponding constructs in the generated object-oriented code, the object-oriented compiler is not able to detect access violations offhand. Hence, we need a feature-oriented compiler with feature-oriented type system. Whereas there are some formalizations of subsets of feature-oriented type systems, there are no fully-fledged compilers that have been developed with feature orientation in mind. Another possibility is to adapt existing compilers of related languages such as CaesarJ [9].

Once we have a feature-oriented compiler, case studies should explore the practicality of feature-oriented modifiers and reveal potential problems but also potential benefits for the mission of attaining real feature modularity.

6. CONCLUSION

Based on our experience with contemporary feature-oriented languages, we have proposed three modifiers targeting specifically at feature-oriented languages mechanisms. Furthermore, we have developed an orthogonal access modifier model that seamlessly integrates object-oriented and feature-oriented modifiers. The background is that the notion of access control has not gained much attention in feature-oriented language design, which leads to a suboptimal modularity and expressiveness and unintuitive semantics and inadvertent errors in feature-oriented programs.

A question that remains is whether the novel modifiers will prove of value in practical software development. Certainly, in order to attain real modularity, further ingredients are necessary (e.g., declarative completeness and modular linking), which are outside the scope of this paper (see the work of Hutchins for details [14]). Also it is not clear whether our names of the modifiers match the intuition of the programmers well. In the case a program element has no modifiers, which modifiers should we assume as default? We intend to initiate a discussion about these and other open issues and inspire further research that evaluates the benefits and drawbacks of our model and its successors.

Furthermore, it is open which further mechanisms are necessary to attain the properties necessary for real modularity (information hiding, modular type checking, and separate compilation) and how they interact with our orthogonal access modifier model.

Acknowledgments

This work is being supported in part by the German Research Foundation (DFG), project number AP 206/2-1 and by the Metop Research Center.

7. REFERENCES

[1] F. Anfurrutia, O. Díaz, and S. Trujillo. On Refining XML Artifacts. In *Proceedings of International Conference on Web Engineering (ICWE)*, volume 4607 of *Lecture Notes in Computer Science*, pages 473–478. Springer-Verlag, 2007.

[2] S. Apel and C. Kästner. An Overview of Feature-Oriented Software Development. *Journal of Object Technology (JOT)*, 8(5):49–84, 2009.

[3] S. Apel, C. Kästner, and D. Batory. Program Refactoring using Functional Aspects. In *Proceedings of the International Conference on Generative Programming and Component Engineering (GPCE)*, pages 161–170. ACM Press, 2008.

[4] S. Apel, C. Kästner, A. Größlinger, and C. Lengauer. Type-Safe Feature-Oriented Product Lines. Technical Report MIP-0909, Department of Informatics and Mathematics, University of Passau, 2009.

[5] S. Apel, C. Kästner, and C. Lengauer. Feature Featherweight Java: A Calculus for Feature-Oriented Programming and Stepwise Refinement. In *Proceedings of the International Conference on Generative Programming and Component Engineering (GPCE)*, pages 101–112. ACM Press, 2008.

[6] S. Apel, C. Kästner, and C. Lengauer. FeatureHouse: Language-Independent, Automated Software Composition. In *Proceedings of the International Conference on Software Engineering (ICSE)*, pages 221–231. IEEE Computer Society, 2009.

[7] S. Apel, T. Leich, M. Rosenmüller, and G. Saake. FeatureC++: On the Symbiosis of Feature-Oriented and Aspect-Oriented Programming. In *Proceedings of the International Conference on Generative Programming and Component Engineering (GPCE)*, volume 3676 of *Lecture Notes in Computer Science*, pages 125–140. Springer-Verlag, 2005.

[8] S. Apel, T. Leich, and G. Saake. Aspectual Feature Modules. *IEEE Transactions on Software Engineering (TSE)*, 34(2):162–180, 2008.

[9] I. Aracic, V. Gasiunas, M. Mezini, and K. Ostermann. An Overview of CaesarJ. *Transactions on Aspect-Oriented Software Development (TAOSD)*, 1(1):135–173, 2006.

[10] D. Batory. Program Refactoring, Program Synthesis, and Model-Driven Development. In *Proceedings of the International Conference on Compiler Construction (CC)*, volume 4420 of *Lecture Notes in Computer Science*, pages 156–171. Springer-Verlag, 2007.

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

[12] P. Clements and L. Northrop. *Software Product Lines: Practices and Patterns*. Addison-Wesley, 2002.

[13] B. Delaware, W. Cook, and D. Batory. A Machine-Checked Model of Safe Composition. In *Proceedings of the International Workshop on Foundations of Aspect-Oriented Languages (FOAL)*, pages 31–35. ACM Press, 2009.

[14] D. Hutchins. *Pure Subtype Systems: A Type Theory For Extensible Software*. PhD thesis, School of Informatics, University of Edinburgh, 2008.

[15] M. Kuhlemann, S. Apel, and T. Leich. Streamlining Feature-Oriented Designs. In *Proceedings of the International Symposium on Software Composition (SC)*, volume 4829 of *Lecture Notes in Computer Science*, pages 168–175. Springer-Verlag, 2007.

[16] R. Lopez-Herrejon, D. Batory, and W. Cook. Evaluating Support for Features in Advanced Modularization Technologies. In *Proceedings of the European Conference on Object-Oriented Programming (ECOOP)*, volume 3586 of *Lecture Notes in Computer Science*, pages 169–194. Springer-Verlag, 2005.

[17] R. Lopez-Herrejon, D. Batory, and C. Lengauer. A Disciplined Approach to Aspect Composition. In *Proceedings of the International Symposium on Partial Evaluation and Semantics-Based Program Manipulation (PEPM)*, pages 68–77. ACM Press, 2006.

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

[19] C. Prehofer. Feature-Oriented Programming: A Fresh Look at Objects. In *Proceedings of the European Conference on Object-Oriented Programming (ECOOP)*, volume 1241 of *Lecture Notes in Computer Science*, pages 419–443. Springer-Verlag, 1997.

[20] T. Reenskaug, E. Andersen, A. Berre, A. Hurlen, A. Landmark, O. Lehne, E. Nordhagen, E. Ness-Ulseth, G. Oftedal, A. Skaar, and P. Stenslet. OORASS: Seamless Support for the Creation and Maintenance of Object-Oriented Systems. *Journal of Object-Oriented Programming (JOOP)*, 5(6):27–41, 1992.

[21] Y. Smaragdakis and D. Batory. Mixin Layers: An Object-Oriented Implementation Technique for Refinements and Collaboration-Based Designs. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 11(2):215–255, 2002.

[22] F. Steimann, T. Pawlitzki, S. Apel, and C. Kästner. Types and Modularity for Implicit Invocation with Implicit Announcement. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 2009.

[23] M. VanHilst and D. Notkin. Using Role Components in Implement Collaboration-based Designs. In *Proceedings of the International Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, pages 359–369. ACM Press, 1996.

Product Derivation for Solution-Driven Product Line Engineering

Christoph Elsner†, Daniel Lohmann‡, Wolfgang Schröder-Preikschat‡
†Siemens AG, Corporate Technology & Research
‡Friedrich-Alexander University Erlangen-Nuremberg
†christoph.elsner.ext@siemens.com, ‡{lohmann,wosch}@cs.fau.de

ABSTRACT

Solution-driven product line engineering is a project business where products are created for each customer individually. Although reuse of results from former projects is widely done, configuration and integration of the results currently is often a manual, time-consuming, and error-prone task and needs considerable knowledge about implementation details.

In this paper, we elaborate and approach the challenges when giving automated support for product derivation (i.e., product configuration and generation) in a large-scale solution-driven product line context. Our PLiC approach resembles the fact that, in practice, the domain of a large product line is divided into sub-domains. A PLiC (product line component) packages all results (configuration, generation, and implementation assets) of a sub-domain and offers interfaces for configuration and generation. With our approach we tackle the challenges of using multiple and different types of configuration models and text files, give support for automated product generation, and integrate feature modeling to support application engineering as an extensive development task.

Categories and Subject Descriptors

D.2.13 [**Software Engineering**]: Reusable Software

General Terms

Design

Keywords

Software Product Line Development, Solution-Driven Software Development, Feature Modeling

1. INTRODUCTION

In classical software product line engineering (SPLE), a software product line is defined as a set of software-intensive systems that share a common, managed set of features satisfying the specific needs of a particular market segment or mission and that are developed from a common set of core assets in a prescribed way [12].

SPLE, however, is not only about *products* in the strict sense, which are usually created according to market needs before being offered to the customer. It covers the realm of software *solutions* as well, meaning that the customer has high influence on the requirements of the software product to develop, which is then created in context of a customer-specific project leveraging reuse.

Feature modeling can guide the whole process of solution-driven product line engineering. In fact, Siemens already uses feature modeling to describe the problem space variability of possible products [18]. It supports scoping of the product line, tendering, cost planning, supports communication between sales&marketing and development, and helps scheduling development releases, testing, and evolution. For documentation purposes, requirements trace into features which in turn trace to the solution space assets. We describe this in further detail in [18].

Although this is far more than using feature models "as a sketch" only, there still remains a gap between the problem space, modeled with features, and the solution space. Several causes currently still hinder using the feature model for product configuration and support partly-automated product generation from the configuration. In particular for solution-driven product lines, where addressing customer-specific requirements and manual implementaion play a central role, an integrated concept for combining automated and manual product derivation is missing.

In this paper, we will (1) describe the characteristics of solution-driven product line engineering and then what we see as state-of-the-art of problem space feature modeling in industry. We (2) derive the challenges for using this feature model for product derivation (i.e., product configuration and generation) in a solution-driven context. We (3) propose and discuss the PLiC approach to tackle the challenges.

2. SOLUTION BUSINESS

Solution-driven business is a project business where results are created for individual customers and their specific problems. It is not the products fitting into a market segment or the optimized production process that is sold to customers. The result, this is the solution or "product" of such projects, is typically only sold once in this form. Examples of solution business are engineering of power plants, production lines, smart homes, hospitals and their building automation, and railway control centers.

FOSD'09, October 6, 2009, Denver, Colorado, USA.

Product Business	Solution Business
n defined products, some explicitly excluded	project business
configuration of fixed set of functions	no product is used twice
foreseeable variability	unanticipated variability
facilitate maintainability of PL platform and products	facilitate/replace copy & paste
"closed world"	open collection of artifacts
example: digital camera	example: power plant, smart home

Table 1: Product vs. Solution Business

Solution business does not mean that every single feature of the result is crafted from scratch. In fact, an efficient solution business is urged to reuse and reapply existing know-how and results to stay competitive. In product business the scope of the product line is fixed by the variability the feature model describes. In solution business, in contrast, each new project reshapes the scope of the product line.

Table 1 compares product and solution business to characterize their differences. The scope of a product line in product business is well defined. It comprises a number of products with a set of common and individual functions. The core asset base is managed, even when application engineering adds functions or when extending the scope of the product line. In solution business, "products" are not clearly defined and show a high variability.

Results in solution business projects share certain characteristics that are determined by the domain. It is therefore useful to manage them together in a software product line. However the different sub-domains of the product line may be rather heterogeneous. They may be covered by standard software *products*, or products derived from other company-internal *product lines*, while others in turn might be developed *from scratch*.

Solution-driven product line engineering basically means that the primary focus is on application engineering, while domain engineering diminishes. In such a scenario, not only the reuse of existing results, but also *product-specific adaptations* and *reactive product line evolution*, that is, when that the concrete applications drive the development of the core product line assets, play very important roles.

3. PROBLEM SPACE FEATURE MODELING

Feature modeling was introduced in [11] as part of the domain analysis and domain modeling phase to systematically describe the common and variable features shared among the products of a product line. A feature model represents the features of a family of systems in the domain and the relationships between them [11]. A valid selection of features from a feature model is called a configuration. Usually, a feature model is considered to be located in the problem space; that is the scope of the analyst who does not care about solution details. This means that the concepts and relations described there are condensed so that they can be understood without detailed implementation knowledge. The actual architectural models and the implementation assets, in contrast, reside in the solution space.

Scoping, in turn, is an analysis activity in domain engineering to find the boundaries of the whole product line, its reusable sub-domains, and its assets [3, 16, 17]. Because there is no agreed definition, we define a sub-domain of a product line as a subpart of the overall product line domain, whereas there is a high cohesion in its problem space, solution space, and in company-internal organization. A sub-domain may be, for example, the operating system, data storage, middleware, GUI frameworks, user management, or domain-specific services. Scoping requires being able to assign business value to decisions on what should be in/out of the product line and carry those decisions on when deciding on what functionality should be built within reusable sub-domains and assets.

Rescoping is necessary to keep the scope of the product line optimized during its evolution. In solution business, rescoping is triggered (at least) for every new project to decide, if a necessary adaptation shall be developed reusable in the context of the overall product line. In this case it must be assigned to an appropriate sub-domain; else it is developed product-specific without following measures for later reuse.

Feature modeling and scoping is a good match. Features are a natural way to describe a domain in terms of problem space concepts [17]. They can be refined and related to solution space assets, and so can carry on business value information. A feature model can, hence, serve as scoping model and define the overall product portfolio and a strategic vision of the variability of the product line. The model is successively refined to map features to concrete sub-domains, and, finally, draw trace links into implementation components of a system. The scoping model includes all common and variable features of a product. The product line architect uses the scoping model to design the reference architecture for all the products.

Once feature modeling is established, it can support various other planning and management tasks. In [18] we describe how it has been used within Siemens to support project and iteration planning and controlling. In general, various task for bridging the "communication gap" between product management, sales&marketing, requirements management, architecting, and development may be supported by the data, like tendering, cost planning, and product line evolution.

In our view, the *essential requirement* on a problem space variability modeling language—and maybe the key success factor of feature modeling—lies in its easy understandability. Involved stakeholders from various backgrounds can immediately grasp the concepts and understand the semantics of feature diagrams, and, at least from the high-level point of view, the expressiveness of (cardinality-based) feature models [6] is sufficient.

However, the downside follows when the real-world solution is to be derived from the asset base. The problem space feature model is far from complete; it only contains high-

level concepts that are not sufficient for complete product configuration. Although there may exist traces from features to the sub-systems and components it affects, *how* to include, exclude, connect, or parameterize a component, in especially, how to generate the actual product, is not part of a problem space model.

Last but not least, variability modeling by only using feature models is quite limited. Constructive variability (instantiation, references) is better expressed in languages that support this concepts directly, and domain experts, for example for business processes, will prefer doing certain configuration tasks using domain-specific languages (DSLs) such as BPEL instead of (mis-)using feature modeling.

4. EXAMPLE SOLUTION-DRIVEN PRODUCT LINE

To illustrate the general problem and the solution proposed in this paper, let us consider the software for a medical digital assistant (MDA) product line for medical hospital personnel. A MDA is basically a personal digital assistant (PDA) with custom software connecting to the hospital information system. For simplification, we will consider two sub-domains: the embedded operating system including the middleware (OS), and the domain-specific business logic including the graphical front-end (GUI). The OS sub-domain is also used in other product lines (e.g., intelligent displays), whereas the GUI sub-domain is only of use in MDAs. The latter needs considerable configuration and manual implementation effort for each hospital, depending on the medical services and workflows it disposes of; equipping a hospital with MDAs is therefore solution business. Both sub-domains have a problem space feature model describing their high-level features. For planning purposes, the MDA product line itself is regarded as a sub-domain and comprises a problem space feature model, which is tied to the two other ones. The OS sub-domain implementation mostly reuses standard software and can be configured via separate text files, whereas the reusable parts of the GUI sub-domain have data and workfow models as configuration input.

5. CHALLENGES OF PRODUCT DERIVATION SUPPORT

Connecting the problem space variability model formally to solution space artifacts for product configuration and generation support is challenging. It requires an unambiguous translation of the concepts of both realms. We identified the following challenges, which, even though they also appear in a product-driven context to a certain extent, are of major importance for efficient product derivation for large-scale, solution-driven product line engineering:

1. Distributed Configuration
   The high-level feature model would become unmanageable if it contained all variability information for all sub-domains. Instead, there should be several variability models to divide and conquer the problem, similar to the hierarchical decomposition of sub-domains of the product line. Constraints between the variability models are necessary to enforce sub-domain–crossing dependencies. This distributed structure also goes in line with staged and multi-level configuration [7], where different stakeholders at different times are responsible for configuring different sub-domains.

2. Heterogeneous Configuration
   The high-level feature model constitutes our link from problem to solution space, and the detailed variability of some sub-domains might be described with feature models as well. However, there are also sub-domains where other forms of configuration are much more suited. Workflow or state machine models describe system behavior, other domain-specific (modeling) languages may describe deployment or replication. Finally, the basic infrastructure often bases on plain text configuration files. Constraints spanning different types of configuration must be possible.

3. Solution Generation Support
   Automating solution creation for those sub-domains that are mature enough to support generation requires connecting the configuration to implementation assets. Heterogeneous types of product generation (e.g., based on models and code generation, compilation, descriptor files, etc.) must be supported as well as a hierarchical mechanism to delegate generation call to all sub-domains.

4. Handling Application *Engineering*
   A new solution may require considerable effort for implementing new features. It is crucial that product-specific features are not neglected, but integrate neatly into the overall configuration and generation process.

Referring to the example in Section 4, the configuration of the MDA sub-domain is hierarchically distributed over the two sub-domains OS and GUI, which have heterogenous types of configuration (text files, models). Generative support only refers to separate sub-parts, and does not allow generating an overall MDA, and the problem space feature models are not used for actual product configuration. There is no sub-domain spanning constraint-checking for validating a configuration, and application engineering is not integrated into the sub-domain configuration and generation process.

6. APPROACHING THE CHALLENGES

In the following, we propose an approach to address the mentioned challenges. We will discuss it in Section 7.

6.1 Product Line Components

In practice, the domain of a product line is divided into sub-domains according to system boundaries and development responsibilities. Our approach encapsulates all artifacts related to a sub-domain—these are *configuration* artifacts, *generation* artifacts, and *implementation* artifacts—into one conceptual entity, a *product line component (PLiC)* (see Figure 1).

Since sub-domains can be hierarchically composed, this is also possible for PLiCs. A PLiC is a development and build-level entity, so interfacing of PLiCs works on the upper two layers: configuration and generation layer. PLiCs are a hierarchical concept, so each PLiC delegates requests for configuration and generation also to child PLiCs. It provides two interfaces: the *configuration interface* and the *generation interface*:

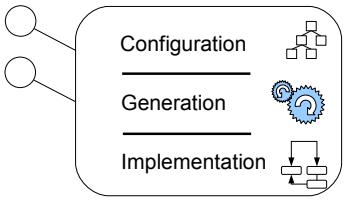


Figure 1: Product Line Component

- Configuration Interface
  - *Configure_PLiC*
    Configuration is a manual task and the interface therefore a human-machine interface. As, in practice, feature modeling alone is often not suited for overall product configuration, we expect that various domain-specific kinds of models or textual configuration languages become necessary. The top level PLiC will be configurable according to the problem space feature model, while the PLiCs for the sub-domains may have arbitrary domain-specific types of models for configuration. Note, that, as PLiCs are hierarchical, this provides support for hierarchical product lines [5], so that a whole (sub-)product line may be a part of the overall solution-driven product line.
  - *Check_Configuration*
    Checking a configuration requires evaluating constraints over all involved options. As these may spread multiple kinds of configuration models and textual files and also may cross sub-domains we need a suitable checking mechanism. Current modeling frameworks, such as EMF [10], fulfill this purpose. All domain-specific meta models developed with EMF correspond to the meta modeling infrastructure ECore. XText [22] facilitates rapid development of parsers for arbitrary textual languages. It outputs a corresponding EMF model for a file written in the language. For other types of models there already exist ECore converters (pure::variants [2] feature models, UML [14]) or can be developed as well. EMF's validation languages (OCL, oAW Check [13]) make building up a *constraint checking infrastructure* over several models and model types feasible.
- Generation Interface
  - *Generate_Solution*
    During application generation the configuration is evaluated and the product is built according to it. For stable sub-domains, the corresponding PLiC may encapsulate a so-called configurable product base [4] so that the product may be derived completely by using generative techniques. For the moment, we regard the generation facilities as a black box, so that arbitrary types of compilation and text and model-based generation and transformation techniques may be used internally.

By bundling configuration and generation facilities directly with the implementation assets, PLiCs approach Challenges 1 to 3. The configuration can be *distributed* over an arbitrary number of models and *heterogeneous* model types and *generation* is carried out hierarchically according to the PLiC hierarchy.

To sum up, our approach implies a hierarchy, where the root PLiC basically contains the problem space feature model to describe variability and global constraints. Referring to our example from Section 4 this would be the MDA problem space feature model and additional constraints. Enforcing correct configuration of the sub-domains conformant to this "abstract" feature model configuration is done via the constraint checking infrastructure. This ensures the configuration of all PLiCs (e.g., the sub-domains OS and GUI) to be valid. Finally, each PLiC generates a sub-product corresponding to its sub-domain, which makes product generation transparent regarding the concrete generation type (e.g., based on model transformations and code generation, pre-processors, etc.). In our example the OS PLiC generates the operating system and the middleware that specifically suite the needs of the domain-specific services and the graphical front-end generated by the GUI PLiC.

6.2 Supporting Solution-driven PLE

To approach Challenge 4, we have to distinguish different types of sub-domains. For solution-driven PLE, certain sub-domains of the solution may be derived from company-internal sub–product-lines, while others are covered by standard software, and others need manual implementation.

6.2.1 Sub–product-lines

Company-internal sub–product-lines fit very nicely into the overall concept. PLiCs have a hierarchical structure, so a hierarchical product line can be built. A sub–product-line PLiC exposes its feature model to the overall feature model. The constraint checking infrastructure ensures a globally valid configuration.

6.2.2 Standard Software

Incorporating standard software (e.g., for infrastructure) into the overall product line is also straight-forward. This means to encapsulate the standard software into a PLiC as well, using the same mechanisms as for sub–product-lines. This way, the detailed configuration of the standard software can be performed in the same manner as for company-internal sub–product-lines, and configuration constraints can be expressed and enforced.

6.2.3 Manual Implementation

In a solution-driven PLE context, manual implementation of assets plays a crucial role. This has both an organizational and a technical facet. The problem space feature model covers the former, the PLiC approach the latter.

Organizational business considerations determine how to implement a new feature's assets. This is where the strength of problem space feature modeling lays. As we indicate in Section 3 and further describe in [18], the decision on how to implement a new feature depends on its attached business values (implementation costs, worth for customer, strategical value, etc.). This technique can both be applied directly to the overall solution feature model as well as be delegated into the feature-models of certain sub–product-lines.

After the organizational decision, if a feature shall be reusable or if not, *technical considerations* come into play. Independently of the decision, the feature may be developed within a PLiC. Each time, it may comprise detailed configuration languages, generation facilities, and the actual implementation. Note, that, although a feature is implemented solution-specific, it will usually still have configuration options for fine-tuning, multiple solution instances, etc. Only its reusability is constrained, as the PLiC makes rigid assumptions about its context, this is the features selected in other sub-domains.

For the actual implementation of variability in a reusable or product-specific way, there exist various possibilities (cf. Table 2). On the one side it is possible to *add*, *change*, and *delete* new artifacts within the reusable asset base. Adding, changing, and deleting is possible on common core assets (C), variation points (VP), and variants (V). On the other side it is possible to *add* new solution-specific variations, to *override* reusable assets, or even to *conceal* common core asset functionality.

	Reusable impl.	Product-specific impl.
Add	Add C, VP, V	Add V
Change	Change C, VP, V	(Override C, VP, V)
Delete	Delete C, VP, V	(Conceal C)

Table 2: Manual Implementation of Assets

Overriding and concealing emerge when there is no suitable variation point in the reusable assets base at a certain location, although one would be needed to perform an adaptation. Usually this is referred to as "patching" and is discouraged; adding an explicit, additional variation point into the reusable asset base and write a variant for it is preferable. This means that, even if a feature shall be implemented product-specific, it still might require adapting the reusable asset base for adding a variation point.

7. DISCUSSION

The PLiC approach is currently in stage of prototypical implementation and evaluation. The interfaces of PLiCs as described above are simple and we have to see if hierarchical composition and black box behavior is sufficient for both configuration and generation. In the following, we discuss the notion of sub-domain, configuration model consistency, scalability, and binding times.

7.1 The Notion of a Sub-domain

The notion of sub-domains is of high relevance in industrial practice of large-scale systems and is used to (hierarchically) structure the overall product line domain. However, there is no generally agreed definition. So, it depends on the context if it refers to problem space, solution space, or organization. In Section 3, we define a sub-domain of a product line as a subpart of the overall product line domain, whereas there is a high cohesion in problem space, solution space, *and* in company-internal organization. Of course, this does not mean there is no interfacing between sub-domains, but that they are relatively stable and clear, in contrast to sub-domain–internal interfacing. This enables us to bundle all artifacts relevant to one sub-domain into a conceptual entity with quite clear interfaces, the PLiC. The problem of partitioning into sub-domains stems from practice and we are convinced that problem space, solution space, and organization have to be considered together to tackle large-scale product line engineering.

7.2 Configuration Model Consistency

From a technical point of view, evaluating the consistency of configuration models is easiest on demand, for example by triggering the evaluation of consistency rules in OCL explicitly. There also exists research about high performance on access/edit constraint checking [9]. The more challenging question is how to manage creation and maintenance of the consistency rules in order to keep the models of heterogeneous types valid. At the moment, we consider using general purpose constraint checking languages, such as OCL, and to assign rule sets to each sub-domain, whereas rules may only access models in their own or in child sub-domains. Checking the consistency of a solution then corresponds to checking the rule set of the root solution-driven product line and all sub-domain rule sets recursively.

7.3 Scalability

The question remains, if such a generic checking infrastructure scales when it comes to large-scale product lines. Next to hard dependencies (requires, excludes), there may be weak ones, for example regarding the influence on execution time or memory usage. These may be covered similar to COVAMOF [19], where weak constraints have attached textual information indicating the impact of a configuration on certain system qualities.

7.4 Binding Times

Having different binding times within the product line could be done via dividing the Check_Configuration service (see Section 6.1) into several services. Each service would represent a binding stage and could introduce stricter constraints. However, we have not elaborated on that issue yet.

8. RELATED WORK

We resemble staged and multi-level configuration of feature models [7] to some extent, as we have a hierarchical model structure that refines from abstract feature to a more and more concrete configuration. However, staged configuration only supports one type of variability model, a feature model. This is the case for many other reported applications, like decision modeling [8], COVAMOF [19], or OVM [15]. In contrast, our approach is free in using various domain-specific modeling languages; we only expect the top level model to be a feature model to link to the problem space.

In contrast to our feature notion, which is rather abstract and driven from problem space considerations, feature-oriented software development (FOSD, [1]) operationalizes a feature as an increment in program functionality that implements a requirement. A sub-domain, as we define it, consists of a set of increments and rules specifying valid combinations (similar to the feature-oriented view on a product line). Ideally, in FOSD, each feature is implemented separately in a so-called feature module to separate the concerns (SoC) of the features. Our approach requires SoC on sub-domain level, while we currently do not explicitly consider SoC within a sub-domain. This means features may be implemented applying strict SoC or not. We aim at implementing SoC on sub-domain-level by classical means, in es-

pecially, providing variation points (e.g., by design patterns) in one sub-domain, while other sub-domains may provide corresponding variants.

Product derivation systems based on version management software [20, 21], manage the reusable asset base (the product line platform) and the reused assets in the actual products separately and preserve dependencies between reusable and reused asset. It follows the assumption that the development of platform and products generally happens independent of each other, but that at some points (e.g., security updates) merges from platform to product assets or vice versa become necessary. In our approach, we try to avoid solution-specific overrides or concealing of assets. Instead, we would prefer to add a variation point to the reusable asset base whenever possible. As, in our application context, the number of concurrent solutions is rather limited (3 to 10) this still seems feasible to us.

9. SUMMARY AND FUTURE WORK

In this paper, we have elaborated and approached the challenges when doing product configuration and generation in a large-scale product line context in solution business. We identified the following four challenges: the need for decentralized configuration, for heterogeneous configuration, for explicit product generation support, and for supporting application engineering as extensive task. We tackle the challenges with the PLiC approach. It resembles the fact that the domain of a large-scale product line is divided into sub-domains. A PLiC (product line component) encapsulates all configuration, generation, and implementation artifacts of a sub-domain. A hierarchical composition of PLiCs then constitutes the overall solution-driven product line. Our approach facilitates decentralized and heterogeneous configuration by allowing multiple models, arbitrary model types, and constraints that may span sub-domains. It gives product generation support by hierarchical delegation of generation calls, and, by integrating a problem space feature model into the derivation process on solution space side, we can give explicit support for application *engineering* as an extensive development process.

Before applying the approach in a real-world context still a lot of research remains to be done. Our next step will be to set up a prototype satisfying the identified characteristics and elaborate on the more technical details of sub-domain–spanning product configuration, consistency checking, and product generation.

Acknowledgments

We thank Christa Schwanninger and Ludger Fiege for their valuable feedback on earlier versions of this paper.

10. REFERENCES

[1] D. Batory. Feature-oriented programming and the AHEAD tool suite. In *Proceedings of the 26th International Conference on Software Engineering (ICSE '04)*, pages 702–703. IEEE Computer Society Press, 2004.

[2] D. Beuche. Variant management with pure::variants. Technical report, pure-systems GmbH, 2006. `http://www.pure-systems.com/fileadmin/downloads/pv-whitepaper-en-04.pdf`, visited 2009-03-26.

[3] J. Bosch. *Design and Use of Software Architectures, Adopting and Evolving a Product Line Approach*. Addison-Wesley, 2000.

[4] J. Bosch. Maturity and evolution in software product lines: Approaches, artefacts and organization. In *Proceedings of the 2nd Software Product Line Conference (SPLC '02)*, pages 257–271, Heidelberg, Germany, 2002. Springer-Verlag.

[5] J. Bosch. Expanding the scope of software product families: Problems and alternative approaches. In C. Hofmeister, I. Crnkovic, and R. Reussner, editors, *Quality of Software Architectures*, Lecture Notes in Computer Science. Springer-Verlag, 2006.

[6] K. Czarnecki, S. Helsen, and U. W. Eisenecker. Formalizing cardinality-based feature models and their specialization. *Software Process: Improvement and Practice*, 10(1):7–29, 2005.

[7] K. Czarnecki, S. Helsen, and U. W. Eisenecker. Staged configuration through specialization and multilevel configuration of feature models. *Software Process: Improvement and Practice*, 10(2):143–169, 2005.

[8] D. Dhungana, R. Rabiser, P. Grünbacher, and T. Neumayer. Integrated tool support for software product line engineering. In *Proceedings of the 22th IEEE International Conference on Automated Software Engineering (ASE '07)*, pages 533–534, New York, NY, USA, 2007. ACM Press.

[9] A. Egyed. Scalable consistency checking between diagrams-the viewintegra approach. In *Proceedings of the 16th IEEE International Conference on Automated Software Engineering (ASE '03)*, Washington, DC, USA, 2001. IEEE Control Systems Magazine.

[10] Eclipse modeling framework homepage. `http://www.eclipse.org/emf/`.

[11] K. Kang, S. Cohen, J. Hess, W. Novak, and S. Peterson. Feature-oriented domain analysis (FODA) feasibility study. Technical report, Carnegie Mellon University, Software Engineering Institute, Pittsburgh, PA, Nov. 1990.

[12] L. Northrop and P. Clements. *Software Product Lines: Practices and Patterns*. Addison-Wesley, 2001.

[13] OpenArchitectureWare homepage. `http://www.openarchitectureware.org/`.

[14] Object Management Group (OMG). Unified modeling language (UML) 2.1.2 superstructure specification. formal/2007-11-02, November 2007.

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

[16] K. Schmid. A comprehensive product line scoping approach and its validation. In *Proceedings of the 24th International Conference on Software Engineering (ICSE '02)*, New York, NY, USA, 2002. ACM Press.

[17] K. Schmid. *Planning Software Reuse - A Disciplined Scoping Approach for Software Product Lines*. PhD thesis, Stuttgart, 2003.

[18] C. Schwanninger, I. Groher, C. Elsner, and M. Lehofer. Variability modelling throughout the product line lifecycle. In *Proceedings of the ACM/IEEE 12th International Conference on Model Driven Engineering Languages and Systems, to appear*. Springer-Verlag, 2009.

[19] M. Sinnema, S. Deelstra, J. Nijhuis, and J. Bosch. COVAMOF: A framework for modeling variability in software product families. In *Proceedings of the 11th Software Product Line Conference (SPLC '07)*, Heidelberg, Germany, 2007. Springer-Verlag.

[20] C. Thao, E. V. Munson, and T. N. Nguyen. Software configuration management for product derivation in software product families. In *Proceedings of the 15th Annual IEEE International Conference and Workshop on the Engineering of Computer Based Systems*, pages 265–274, Washington, DC, USA, 2008. IEEE Control Systems Magazine.

[21] J. van Gurp and C. Prehofer. Version management tools as a basis for integrating product derivation and software product families. In *Proceedings of the Workshop on Variability Management - Working with Variability Mechanisms at SPLC 2006*, pages 48–58. Fraunhofer IESE, 2006.

[22] Eclipse XText homepage. `http://www.eclipse.org/Xtext/`.

A Model-Based Representation of Configuration Knowledge

Jorge Barreiros[1,2]
[1]Inst. Superior de Eng.de Coimbra;
Inst. Politécnico de Coimbra
Coimbra, Portugal

jmsousa@isec.pt

Ana Moreira[2]
[2]Departamento de Informática
Faculdade de Ciência e Tecnologia
Univ. Nova de Lisboa

amm@di.fct.unl.pt

ABSTRACT
Implementation of feature-oriented systems is typically made by creating an admissible configuration, according to a specified feature diagram, that dictates what artifacts are to be composed to create the desired solution. These artefacts are typically grouped according to the feature they concern. However, some artefacts may be related not to a specific feature, but to a combination of them. Also, multiple alternate implementations of a single feature may exist, and the preferred one may be dependent on the specific configuration that is being composed. We propose a graphic model to represent configuration knowledge that is able to address such concerns.

Categories and Subject Descriptors
D.2.2 [**Design Tools and Techniques**]

General Terms
Design, Languages

Keywords
Feature Modeling, Knowledge Representation Modeling, Feature Interaction, Feature Dependency

1 INTRODUCTION
Techniques for deriving low level design models or implementation from a feature diagram usually assume a more or less direct correspondence from a feature into a single design artifact or implementation (we use the term "feature implementation" loosely, referring to either low level design models or other development artifacts, such as source code files).. This is unfortunate, because multiple candidate implementations may compete to implement a given feature. In this scenario, the selection of the actual implementation to be used for a feature is based on context specific rules, restrictions and optimization criteria. Rather than requiring the developer to expand the feature diagram with unnecessary detail, Software Product Line tools such as Pure::Variants [1] handle this problem by allowing the developer to specify configuration rules that determine what composition artifacts are to be generated. This promotes a clear domain separation, that is, the representation of concepts deriving from domain analysis is not unnecessarily tangled with concepts concerned with the implementation of the system.

We propose to explicitly represent this configuration knowledge. using a graphic model technique that is suited to represent design and implementation knowledge in a combined diagram.

The remaining of this paper is organized four sections. In Section 2, we will discuss the background and motivation for our work. In Section 3, we present our model with the help of a mobile media application case study. In Section 4, we discuss related work and we conclude this paper in Section 5.

2 BACKGROUND AND MOTIVATION
Techniques such as the ones described in [2-4] have been proposed to address the mapping from feature to its implementation. In all these cases, a one-to-one mapping from feature to implementation is generally assumed to exist, even though the adoption of aspect-oriented techniques may be used to create feature implementations or designs that crosscut the entire implementation space. But even in this last case, a single aspect is assumed to implement the feature. Significant flexibility could be gained by allowing the dynamic selection of one among multiple implementation alternatives.

A related problem is that it can be hard to cleanly distribute implementation artefacts among isolated features. In fact, some of these may be of relevance only when two or more individual features are selected, making no sense to assign them to a single, isolated feature.

Finally, in some cases, implementation artefacts may need to be customized according to the desired configuration. For example, if aspectual model components such as the ones described in [5, 6] are used, then it is necessary to provide the proper instantiation parameters required according to the specific configuration. Some source file transformation might also be necessary, driven by the specific needs of the selected configuration.

By creating a model to represent specifically this configuration knowledge (and relevant dependencies and interactions between

FOSD'09, October 6, 2009 Denver, Colorado, USA

features), we are able to properly represent and address all these issues. Our model can be added to existing tools as an alternate view for displaying and editing the configuration knowledge of a Software Product Line .

3 A MODEL FOR CONFIGURATION KNOWLEDGE REPRESENTATION

3.1. Multimedia Application Case Study

The feature model in Fig. 1, describing a family of multimedia applications for mobile devices, will be used to illustrate our approach. This example is partially based on work presented in [7, 8]. The following restrictions and dependencies apply:

- The "Send Photo" feature is dependent on the inclusion of the "Photo" feature.
- If both "Photo" and "Video" media options are selected then an additional menu must be included to allow for the user to switch between both media types. If only one media is present, no such functionality is required.
- For achieving acceptable performance, a 3rd party video decoder must be used in systems with higher resolution. In lower resolution systems, an in-house solution was found to be acceptable and is to be used instead, to reduce royalty costs.

These configuration restrictions, as we will show, are easily represented in our configuration knowledge graphical model.

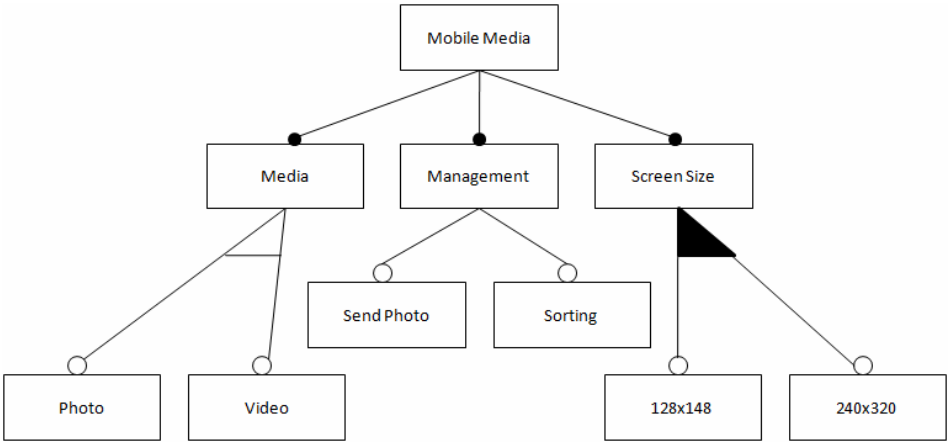


Fig. 1 - Mobile media application case study

3.2. Configuration Modules

Feature implementations are described using configuration modules. A generic configuration module is shown in Fig. 2.

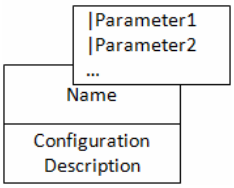


Fig. 2-Configuration Module

Name is an optional tag that names the configuration module. It will typically match the feature name if the configuration module, describing the configuration of a specific feature, but that may not always be the case. An optional list of parameters is also present. Each parameter will be used to tailor the described configuration. Lastly, the configuration description itself will be presented. The specific details of this description depend on the desired type of implementation artefacts we are dealing with. A typical application will use aspectual model components if the output of a detailed design model is desired. Alternatively, if a makefile or similar project description is the desired output, the names of the relevant source files can be used instead. We will adopt the later approach for the examples given in this text.

The configuration module in Fig. 3 describes the configuration properties for the mobile media feature in our case study. As we can see, the configuration parameters are those necessary to completely specify a mobile multimedia application. Our model will describe how this information is to be propagated to other configuration modules to properly create the system. The file *mobileMedia.java* is referred to in the configuration description. This file will be included in every multimedia application project that is created by configuring the Mobile Media feature.

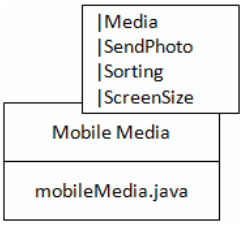


Fig. 3 - Configuration Module Example

From the perspective of the developer, all that is required to create a new mobile media family member will be to properly configure the "Mobile Media" feature by supplying the requested parameters. The configuration knowledge diagram may then be processed to take care of the additional configurations required to implement all relevant features.

3.3. Associations and Configuration Propagation

Configuration modules can be associated to represent feature and configuration dependencies. Dependencies can be checked and validated using a cardinality based approach [9] (cardinality 0 can be used to represent exclusion). These associations can also be used to describe how configuration information is to be propagated across the configuration modules.

As an example, consider Fig. 4, where an association is represented between the "Mobile Media" and "Media" configuration modules. This association describes two important properties:

- 1 or 2 distinct configurations of "Media" are required per each configuration of "Mobile Media",
- The |MediaType parameter of the Media configuration module is to be initialized by using information supplied in the "|Media" parameter of the "Mobile Media".

The |Media parameter of the "Mobile Media" configuration module may be set to a list of multiple values. Rather than propagating this list itself, the asterisk in the |Media* reference in the association indicates that we are referring to each and every one of the list values. So, if |Media={Photo, Video}, for example, then two distinct configurations of the "Media" configuration module are created with |MediaType="Photo" and |MediaType ="Video", rather than a single configuration with |MediaType = {Photo,Video}.

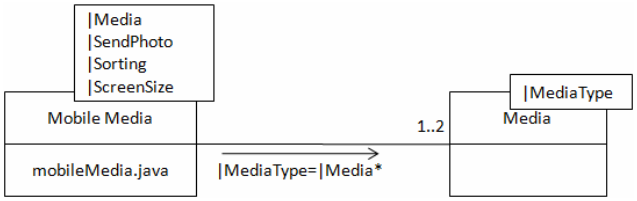


Fig. 4 - Association between configuration modules

Composition can be used as a shorthand notation for indicating that similarly named parameters on either side of the association should be set to the same value. A guard can also be used to indicate under what circumstances an associated configuration module should also be instanced. These applications of associations are represented in Fig. 5

3.4. Configuration Module Specialization

No configuration description is represented in the Media configuration module in Fig. 4. This is deliberated, as the mechanism of configuration knowledge specialization is used instead to represent the alternative implementations of the photo and video sub-features. This mechanism will also be used to select from codec implementations, according to the stated restrictions of this product family.

A specialization of a configuration module is a specific implementation of that configuration module that is to be used whenever a specific value or set of values is used in the parameters.

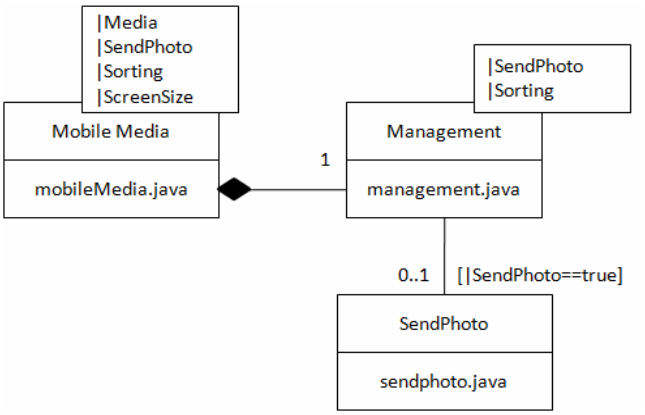


Fig. 5 - Composition and dependencies

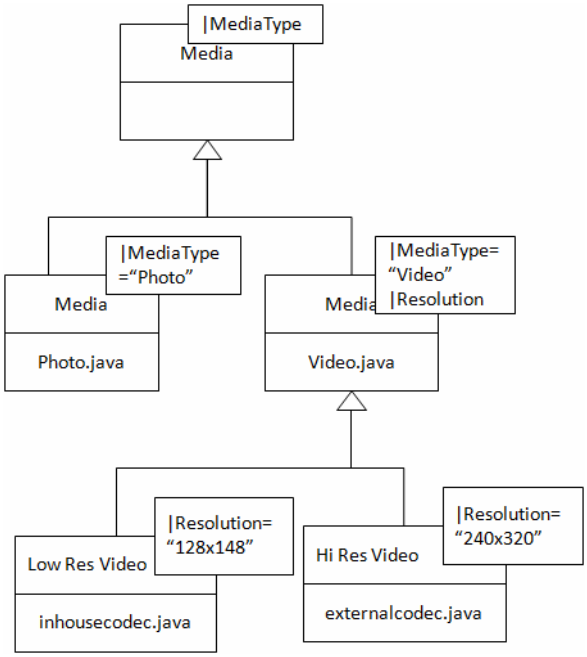


Fig. 6 - Configuration Module Specialization

In the example of Fig. 6, specific variants of the "Media" configuration module are provided for each one of the special cases represented by having the |MediaType parameter set to either "Photo" or "Video". Further specializations of the "Video" configuration module are used to handle the alternative codec implementations, depending on the value of the |Resolution parameter. This value can be propagated from the "Mobile Media" configuration module in a similar manner as described in Section 3.3.

It should be noted that since the photo media configuration is reified, it is possible to cleanly represent the dependency between the "Photo" and "Send Photo" features using an association, as shown in Fig. 7 .

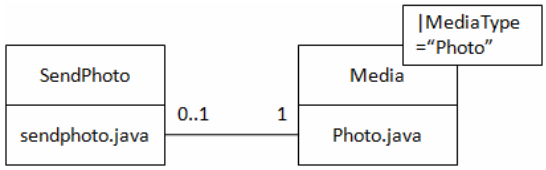


Fig. 7 – Representation of the dependency between SendPhoto and Photo.

3.5. Multiple Feature Interactions

According to the case study description in Section 3.1, a menu should be included to allow the user to switch media types if both the "Video" and "Photo" features are selected. This does not map cleanly to any isolated feature of our diagram, since it is not actually depending on either the "Photo" or "Video" features when considered in isolation. We represent this in our model by allowing *N-ary* associations among multiple configuration units.

For the situation described in our case study, the diagram in Fig. 8 is adequate. This diagram should be interpreted as follows: if either the "Photo" or "Video" are included, and if they are included in number greater than 1, then apply the configuration module that adds the option menu (we have chosen to deliberately represent it as an anonymous configuration module to emphasize it does not relate specifically to any feature).

This configuration could be easily extended to support additional configurations like illustrated in Fig. 9 and Fig. 10.

In Fig. 9, we represent alternate configuration choices between a toggle menu (if only two media options are selected) and a list menu (if more than two media options are selected), in a hypothetical scenario where a third Media option was considered ("Audio" in this example).

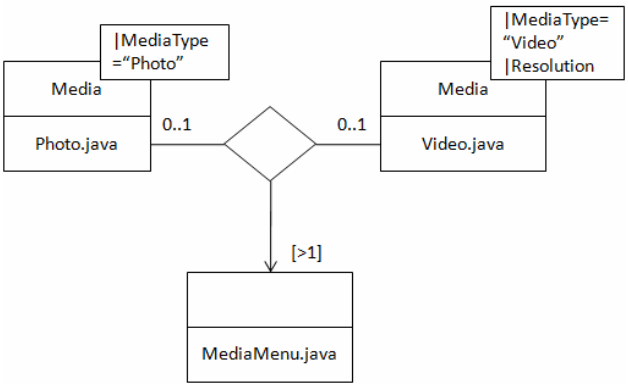


Fig. 8 - Using an N-ary association to include an option menu.

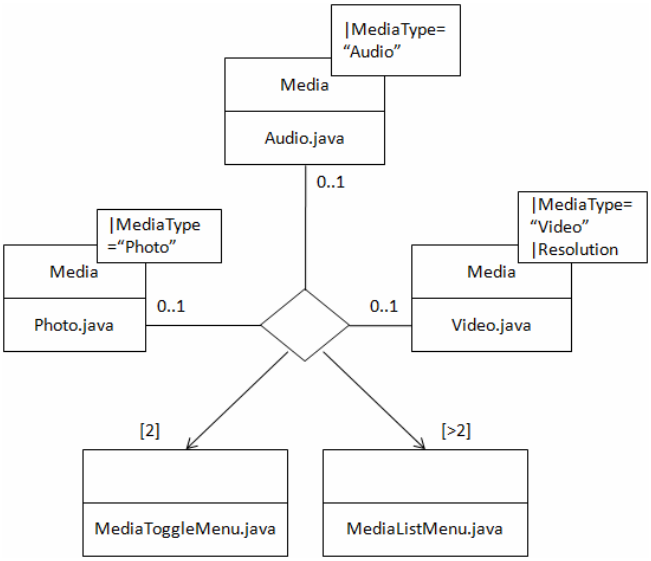


Fig. 9 – Handling multiple menu options, depending on number of selected features.

In Fig. 10, the following situation is described: if the "Edit" feature is selected (notice the cardinality 1) and at least one of "Audio" or "Video" are selected, then a menu for audio pitch control should be created. A similar pattern is used for the "Edit", "Photo" and "Video" features, where a Color edition menu is selected instead. This example shows how multiple, complex dependencies can be efficiently represented in our model.

3.6. Wrapping up

For the sake of completeness, we present the remaining configuration knowledge diagram for our case study, even though we reuse the mechanisms already described. The different screen

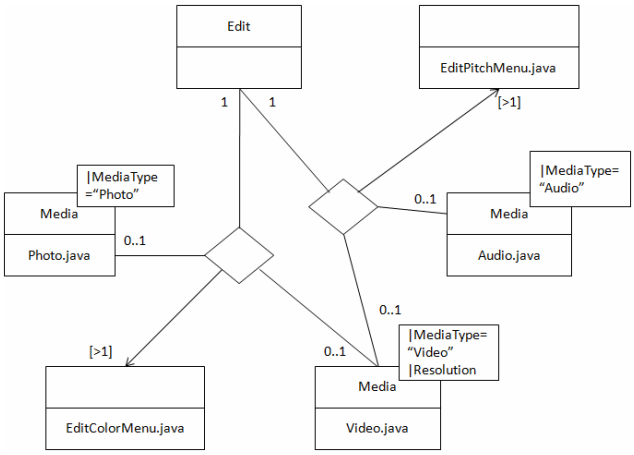


Fig. 10 – Another example of multiple configuration options depending on feature selection

sizes are considered to be implemented by two AspectJ modules that will be responsible for making the necessary adjustments across all implementation artefacts that somehow depend directly on screen size resolution to function properly.

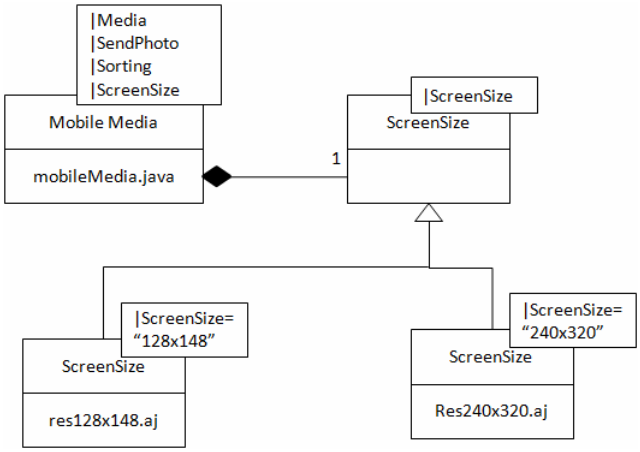


Fig. 11 – Screen size configuration model

In Fig. 11, the "Screen Size" configuration module is initialized with the propagated homonymous parameter from the "Mobile Media " configuration module. The two available resolutions are implemented by two appropriate specializations.

4 RELATED WORK

Many different works address issues related to configuration knowledge management, feature interaction, variability modeling and feature implementation. Our work is able to address representation of design or implementation variability, while offering constructs for representing feature dependencies (N-ary associations), interactions (through cardinality restrictions) and variable implementations (through specializations).

Several techniques have been proposed for implementing features. These techniques differ on the kind of generated artefacts (source, design models, etc) and on the specific approach to compose these assets. As representative examples, we can consider [2-4]. *Mixins* are used in the AHEAD framework [2] to compose classes based on selected features. In [3], a template-based model of all superimposed variants is used to create a design model of the configured product. In [4], Jayaraman et. al. propose the use of the MATA framework for creating aspectual model slices for each feature that are subsequently composed into an appropriate design model according to the selected configuration. Our model approach is more generic in the sense that it is used to describe what assets are to be composed, but makes no specific restriction on the kind of artefacts and composition method that is subsequently employed. Also, variable feature implementations are not directly supported in any one of these approaches: a single correspondence from a feature to an implementation artifact (mixin, model slice or a single superimposed design model) is assumed. Software Product Lines tools such as Pure::Variants [1] superimpose additional configuration management over such feature implementation techniques to achieve variable feature implementation. Our work allows such information to be represented and edited through a model that represents configurable variability both in solution and problem space..

The orthogonal variability model described in [10] is able to partially achieve a similar effect, by using feature diagrams to externally represent variability across different views. However, only dependencies and interactions between single features are considered, which is problematic for representing cases such as the one in Fig. 10..

Configuration modules are customizable by use of generic parameters. Depending on the specific composition method that is used, these parameters may be used to allow for additional configuration of the selected artefacts. For example, if RAM or RMS aspectual model components [5, 6] are used, then the parameters of each aspectual component may be initialized from values supplied through the configuration module parameters.

5 CONCLUSIONS AND FUTURE WORK

We have shown an example application of our modeling technique to describe the configuration details for a mobile multimedia application. Our configuration knowledge model has managed to capture the relevant details necessary for successfully configuring a family member according to the presented feature restrictions and interactions.

Although in our example we have used our model to select the correct files that should be selected for building the configured application, our work is easily generalized to other implementation options such as *mixin* selection or detailed design modeling composition through aspectual modeling techniques.

As future work, implementation and automated integration in tool chains should be conducted. Definition of an appropriate meta-model should be conducted, and the technique should be combined with aspectual modeling techniques to offer a concrete solution, applicable in Model Driven Development scenarios. Applicability to large scale real life scenarios should also warrant further consideration.

ACKNOWLEDGMENTS

This work has been partially supported by FCT grant SFRH/BD/38808/2007 and by the European FP7 STREP project AMPLE [11]

REFERENCES

[1] "Pure::Variants (http://www.pure-systems.com) ".

[2] D. Batory, "Feature-Oriented Programming and the AHEAD Tool Suite," in *26th International Conference on Software Engineering*, 2004.

[3] K. Czarnecki and M. Antkiewicz, "Mapping Features to Models: A Template Approach Based on Superimposed Variants," in *4th International Conference on Generative Programming and Component Engineering (GPCE)* Tallin, Estonia, 2005.

[4] P. Jayaraman, J. Whittle, A. M. Elkhodary, and H. Gomaa, "Model Composition in Product Lines and Feature Interaction Detection using Critical Pair Analysis " in *10th International Conf. on Model Driven Engineering Languages and Systems (MoDELS 2007)* Nashville, 2007.

[5] J. Kienzle, W. A. Abed, and J. Klein, "Aspect-Oriented Multi-View Modeling," in *AOSD'09* Charlottesville, Virginia, USA, 2009.

[6] J. Barreiros and A. Moreira, "Reusable Model Slices," in *14th International Workshop on Aspect-Oriented Modeling* Denver, 2009.

[7] P. Borba, "Software Product Line Refactoring tutorial," in *GTTSE'09 Summer School* Braga, Portugal, 2009.

[8] E. Figueiredo, N. Cacho, C. SantíAnna, M. Monteiro, U. Kulesza, A. Garcia, S. Soares, F. Ferrari, S. Kan, and F. Filho, "Evolving software product lines with aspects: An empirical study on design stability," in *ICSE* New York, USA, 2008.

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

[10] K. Pohl, G. Böckle, and F. v. d. Linden, *Software Product Line Engineering*: Springer, 2005.

[11] "AMPLE Project Webpage (www.ample-project.net)," 2009.

Domain analysis on an Electronic Health Records System

Xiaocheng Ge
Department of Computer Science
University of York
York, United Kingdom
xchge@cs.york.ac.uk

Richard F. Paige
Department of Computer Science
University of York
York, United Kingdom
paige@cs.york.ac.uk

John A. McDermid
Department of Computer Science
University of York
York, United Kingdom
jam@cs.york.ac.uk

ABSTRACT

Electronic Health Records (EHR) have been proposed as a means for managing the technical and organisational complexity that arises in modern healthcare. Different EHR systems are being developed around the world, and within individual countries, different services, such as electronic prescriptions, are being deployed that exploit EHR. We report on a domain analysis of England's developing EHR, as is being implemented in the NHS's National Programme for IT. The analysis, supported by the Feature-Oriented Domain Analysis (FODA) process, ultimately aims to identify commonality and variability across services that use EHR. We summarise the analysis, and describe challenges that we encountered when using FODA to support the analysis.

Categories and Subject Descriptors

H.2 [**Database Management**]: Systems; D.2.1 [**Software Engineering**]: Requirements/Specification—*Methodologies*

General Terms

Design, Experimentation

Keywords

Feature Oriented Domain Analysis, Healthcare Domain, Electronic Health Record

1. INTRODUCTION

The demand for high-quality healthcare continues to rise. The care provided today is much more complex, both technically and organisationally, than ever before. One area in which technical complexity has increased is in recording and storing patient *health records*. Paper-based health records have been in existence for centuries; their gradual replacement by electronic records has been proceeding for the last twenty years, in many different countries [8, 12]. Governments in Australia, Canada, Denmark, Finland, France, New Zealand, England and the United States have announced and are implementing plans to build integrated computer-based national healthcare infrastructures, based around the deployment of interoperable electronic health record (EHR) systems. Compared with paper-based systems, EHR-based systems are supposed to improve the safety and quality of patients' healthcare. However it has been demonstrated that implementing such large-scale healthcare IT programmes can be extremely difficult and costly [1, 5, 13, 14].

England's National Programme for IT (NPfIT) was initiated in 2003 to deliver a number of major initiatives that will enable an EHR system. The system is to be based on new network infrastructure, as well as national (standardised) services that will be based on the EHR (e.g., for supporting electronic transfer of prescriptions, and electronic booking of appointments). NPfIT is the largest outsourced public-sector IT project [10], and its size and complexity has led to substantial concerns being raised over progress towards deployment [1].

The national implementation of EHR in England is continuing, and the requirements for EHR and *applications* of EHR are continually changing. In parallel, we are investigating the challenges of implementing parts of the NPfIT programme from a software engineering point of view. In particular, there are other substantial EHR programmes throughout the world, and understanding similarities and differences between these programmes may provide benefits to England's programme, through improved understanding of technical challenges and solutions. Additionally, as mentioned above, there are a number of services that are making use of, or are intending to make use of, EHR, and the requirements for these services is changing. There is likely to be substantial commonality, as well as points of variation, amongst the functionality provided by these services, and in how the services make use of EHR.

The overall aim of our recent research has been to attempt to understand these *commonalities* and *variations* across services and EHR programmes, and to use this to help improve existing EHR infrastructure.

Our basis for this work has been the application of feature-based software engineering, particularly techniques developed for software product lines and software architecture. In this, we effectively treat EHR-based software systems as software product lines. The key idea that we take from these areas is the development of products from reusable core assets [3, 7, 9]. To develop core assets, understanding the commonality and variability (C&V) between the prod-

FOSD'09, October 6, 2009, Denver, Colorado, USA.

ucts in the domain is essential [11, 15, 16]; feature-based approaches have been used extensively to support this [7, 15, 16].

The first stage of our ongoing research is to use feature-based approaches to analyse the commonality and variability of the domain of EHR systems in England, focusing on existing NHS NPfIT services (including the EHR system in England, electronic appointment booking, and electronic prescriptions). Our view is that an analysis of C&V may help in promoting understanding of shared requirements, simplifications to software architecture and database architecture, and may help to accelerate implementation and deployment. Ultimately, we are aiming to compare existing architectures of EHR systems with those that can be designed using feature-based approaches, in order to help to identify improvements that may be made to these existing EHR systems.

This paper presents a domain analysis of NPfIT's EHR, based on an application of Feature-Oriented Domain Analysis (FODA) [17], and as a result makes two contributions. Firstly, we summarise the results of a FODA analysis that takes into account a number of sources of domain information, including relevant standards for EHR, service specifications, and NPfIT policy documents. This is the first step towards a more detailed FODA-based C&V analysis of designs and implementations of EHR systems around the world, which will ultimately help to derive improvements to existing EHR architectures. Secondly, we will describe open problems and challenges that we identified during the application of FODA to this domain, with the intent of proposing changes to the analysis approach.

In the next two sections, we will briefly introduce EHR and EHR systems, and will then describe the architecture of existing EHR systems within the NPfIT. Then, we will briefly describe the FODA process we applied, and the outcomes of the analysis, as applied to the NPfIT EHR systems. Finally, we will summarise open questions and problems that were identified as a result of the analysis.

2. EHR AND EHR SYSTEMS

The phrase 'electronic health record' is generally used to describe any digital representation of health information, with little or no concern about how this information is to be stored or retrieved. Terms used approximately synonymously include electronic medical record (EMR), electronic patient record (EPR), electronic health record (EHR), and computer-based patient record (CPR). In general, these terms can be used interchangeably, but some specific differences can be identified. For example, an Electronic Health Record typically refers to a longitudinal record of a patient's care carried out across *different* institutions and sectors, whereas an Electronic Patient Record has been defined as encapsulating a record of care provided at a *single* site (e.g., by a single health trust). In England, the NHS uses the term EHR to describe the concept of a longitudinal record of patient's health and health care, from cradle to grave. It combines both information about patient contacts with primary health care (such as a doctor or a hospital), as well as subsets of information associated with the outcomes of periodic care, which are also held in EPRs.

Different organisations define and structure EHRs in different ways, illustrating different intended applications of use. For example, the definitions from HINA (Australia) and OHIH (Canada) emphasise the characteristics (or properties) obeyed by EHR; the definitions from ASTM (USA) and CPRI (USA) highlight the contents of EHR, and IOM's (USA) definition focused on the objectives of EHR. There is no agreement or standard definition for EHR, nor is there a suitable definition of what constitutes a necessary or sufficient EHR. For the purposes of our research, it is useful to de-emphasise the focus on what systems are required in a given care context (e.g., primary care, community care, or mental health). Instead, it is helpful to look generically at the information that arises during management of patient health, regardless of the systems that generate or contain it, and to generically view the systems that create, manage and store EHRs in the context of *jurisdictional information sharing* [4].

For these reasons, we have adopted the ISO definition of EHR. In [2], EHR is defined thusly: *"The basic generic definition for the EHR is a repository of information regarding the health status of a subject of care, in computer processable form."* The IS definition further acknowledges that the sharing of EHR information can take place at three levels:

Level 1 - between different clinical disciplines or other users, all of whom may be using the same application, requiring different or ad-hoc organisation of EHRs,

Level 2 - between different applications at a single EHR node - i.e., at a particular location where the EHR is stored and maintained, and

Level 3 - across different EHR nodes - i.e., across different EHR locations and/or different EHR systems. And when level 3 is achieved and the object of the EHR is to support the integrated care of patients across and between health enterprises, it is called an Integrated Care EHR (ICEHR).

More generically, Level 1 involves sharing within one organisation and application, but by different users; Level 2 involves sharing within one organisation, but across different applications and users; finally, Level 3 involves sharing across different organisations, applications and users.

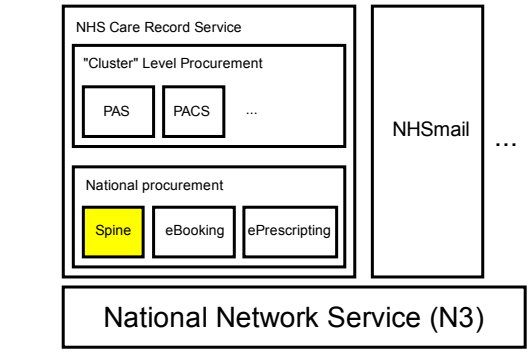
In addition, the ISO definition of EHR distinguishes between clinical information and the systems that support its provision. What we are interested in is the analysis, design and implementation of a national EHR and the applications that surround it. Thus, the term 'EHR system' in this paper refers to the information systems that stores and processes the data of EHRs; in its simplest form, this could be a database application.

3. ARCHITECTURE OF NHS EHR

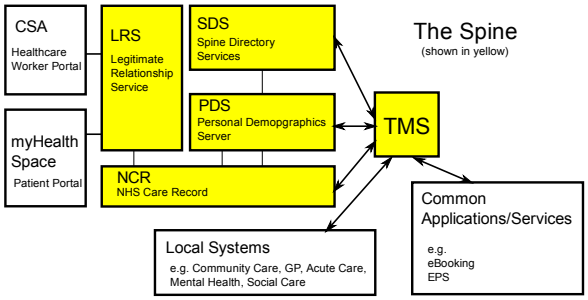
In this section, we give an overview of the current architecture of the NPfIT's EHR systems for England. This architecture has evolved over a number of years, as described in [1].

There is some compatibility between the ISO definition of EHR [2] and the architecture used in England, e.g., the separation of EHR systems into *local* EHR systems (used by one care provider) and *shared* systems. In England, the EHR system was created at two levels: a Summary Care Record (SCR) accessible across England; and a Detailed Care Record (DCR) accessible within a locally determined health community which could encompass primary

and secondary care providers within a specified geographical area, e.g., London. In both cases, access to the EHRs will be subject to stringent confidentiality and security controls. The SCR primarily supports 'out-of-hours' and accident and emergency care and will eventually provide a 'cradle to grave' record of significant health information. The DCR supports more routine health interventions and will (eventually) replace organisationally based record systems such as those of hospitals and GP practices.



(a) The Spine in the Architecture of NPfIT



(b) Architecture of the Spine

Figure 1: The Scope and Structure of EHR in England

Figure 1(a) shows the national care record service in the architecture of entire NPfIT programme. The key features of this programme are new national data and IT standards, procured and paid for nationally. Implementation in acute trusts will be through one of five geographic partnerships with industry, called "clusters". The foundational infrastructure is a New National Networking service (N3). Based on this N3 service, there are several national services, e.g., electronic booking (called "choose and book"); electronic transfer of prescriptions; and a nationally accessible, life-long summary patient record called "the spine". The provision of electronic functions at local level form part of the NHS care record service, a collective term for all aspects of clinical IT support applications, from clinical decision making tools to digital X rays. The output of those applications is intended to be a health record that can be shared. Figure 1(b) is the architecture of the Spine. The spine consists of the following major components:

- TMS: transaction and messaging spine; the master "router" of all messages between systems. All messaging via the TMS is based on the HL7 version 3[1] Clinical Data Architecture (CDA).
- SDS: Spine Directory Services; provides various Directory Services (e.g., organisational details of GP practices). SDS excludes patient related demographics.
- PDS: Personal Demographics Services; provides a national service holding all personal, demographic and related information for each patient.
- NCR: National Care Record; holding a summary of clinical and associated information.
- LRS: Legitimate Relationship Service; controls what access a healthcare professional has to a person's clinical data.

4. A DOMAIN ANALYSIS OF AN EHR SYSTEM

In the previous section we described the existing EHR systems for England's NPfIT. In this section we summarise a domain analysis for an *idealised* EHR System in the context of the NPfIT, based on a feature model. We ultimately intend to use the results of the domain analysis to help develop an architecture for an idealised EHR system, which can thereafter be compared with the *existing* EHR systems presented in the previous section. However, the purpose of this paper is to describe some of the challenges we encountered in applying FODA to this domain.

To set about this, we must first more carefully scope what is in context, and what is not. Conceptually, an EHR system can be treated as a database application which stores and manages the data of patient health records. More concretely, an EHR system, according to key reports such as [6], provides eight core functions: storing health information and data; managing results; managing orders (e.g., for prescriptions); supporting decision making; improving connectivity; raising patient involvement; managing administrative processes; and reporting.

Additionally, other information constraints, requirements and functions can be identified from key literature related to NPfIT. In particular, the report on the NHS logical health record architecture (LHRA) [18] indicates that the LHRA programme is intended to provide a basis for clinical and informatics communities to achieve widespread sharing and use of clinical information via EHRs. This document captures overall requirements for a NHS LHRA in terms of current and future needs; however, these requirements may not be the same as those for NHS EHRs. Nevertheless, the future requirements for the LHRA help us to identify additional features that may be of use in the EHR programme.

Based on the description of LHRA and the report in [6], we produced a feature model that captures these additional features. A more detailed description of the features is in the appendix.

In the list, the top level features are *F1 store the clinical data*, *F2 process the data*, *F3 decision support*, *F4 administrative management* and *F5 patient support*. These are the key features listed in [6] and they are implemented in almost every EHS system around the world. Considering England, the requirements which are not supported by current system

[1] `http://www.hl7.org/`

and desired by future needs are documented in [18]. In the feature model (Figure **??**), the existing features are omitted. In the diagram, they are new features under the key features. The description of these new features can be found in Appendix.

We used the results of applying FODA to produce a prototypical architecture for an EHR, shown in Figure 2, which we aimed to use as a basis for comparison with existing EHR systems. The FODA results led us to choose between a service-oriented architecture and model-view-controller architecture. More specifically, we determined that there would need the following components:

- Interface to the public, which is a portal to the public. Its function will be to provide educational meterials to the public so that it will increase the awareness of national health.
- Interface to patients, which is a portal to the patients. It achieves the feature of *patient support* — feature *F5* and its sub-features in Appendix.
- Authentication and access control, which are basic security mechanisms of the system. Its function is to provide an authentication service and control access from outside of system.
- Medical data processing service, which is the *"model"* of a MVC architecture. It processes the patient's medical data, which corresponds to feature *F2*.
- Patient summary and historical data services, which are data storing services. The patient summary data service stores the summary and the data regarding the patient's current health status, while the patient historical data service stores a summary of the patient's health history. These two services are the kernel of entire system, which support most of the features listed in the appendix.
- NHS service directory, which provides a directory service. Because the NHS EHR system is a distributed system with a service-oriented architecture, this service provides information which links to other NHS services or local systems when they are needed by the service of data processing or local healthcare system.
- Interface to other NHS services and local healthcare system, which will provide an interface to other systems. Together with data storing services, they support the features of data sharing and processing, e.g., *F21, F31* and *F53a*.

The intention of this exercise was to be able to compare the results of FODA and a prototypical architecture with an *existing* architecture. In the next section we outline some key open questions that result from comparing it with our results.

5. OPEN QUESTIONS

We followed the process of FODA, as described in [17] - content analysis, domain modelling, and architecture modelling - to analyse the domain of EHR systems in the NPfIT. During the process, we identified several challenges and open questions related to FODA. We briefly describe these here.

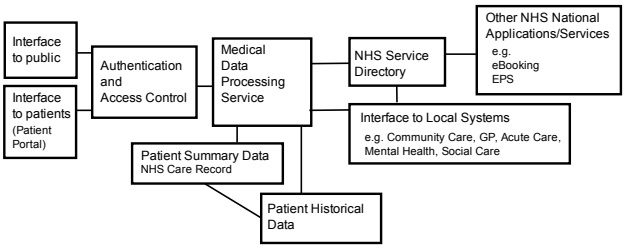


Figure 2: Prototypical Architecture of NHS EHR in England

5.1 Architectural Concerns when Analysing Content of System

Most applications of domain analysis techniques, including FODA, focus on non-distributed systems; our domain of interest is large-scale distributed systems. There are many difficulties and challenges related to applying FODA (and similar techniques) to such systems. One such challenge is that it can be complicated to precisely identify the boundary of such a system, particularly large-scale distributed systems based on a loosely coupled service-oriented architecture. There are other challenges as well, particularly related to the extent to which system *architecture* should be considered with the FODA process, as we now explain.

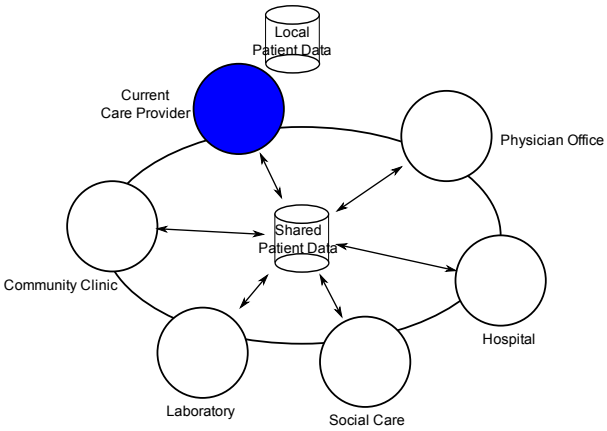


Figure 3: Deployment Architecture of NPfIT Spine

There are many architectural styles that have been used for large-scale distributed systems; [12] summarises four popular ones. For example, the EHR system in England has a centralised architecture, illustrated in Figure 3. In the case of EHR systems (and other, similar systems), the choice of architectural style directly dictates the feature set. Hence, the architectural style should be considered when the context of the EHR system is analysed. More specifically, the system architecture for EHR systems is both a result of domain analysis and (at least partly) a prerequisite of domain analysis. Consider the example of Feature 1 in List 6: *storing clinical data*. This feature needs additional features in order to synchronise the database operation, in the case where the data is stored in a distributed manner. In practice, it is likely that only a partial view of the architecture will be needed to help control the process of domain

analysis.

Our perspective is that the domain analysis for distributed system should be carried out iteratively, e.g., via a loop added to the domain analysis process suggested in [17]. Our suggested process is shown in Figure 5.1.

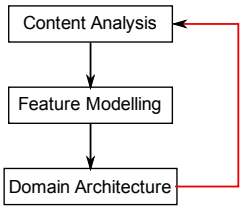


Figure 4: Iterative Domain Analysis

In practice, it is not difficult (and sometime it is necessary) to refine the understanding of the system, its scale and scope. The question is to which degree the details of the system architecture should be considered in order to maintain a cost-effective analysis process. This question seems to be open because the answers of this question may depend on the domain of the application. In the domain of EHR systems, a generic domain model may not be as useful as expected because variations between individual EHR system and the generic model may be too substantial.

5.2 Organisational Goals vs. Feature Model

Political objectives and policy are often one of driving forces behind the implementation of EHR systems. In initial proposals and requests for expressions of interest for implementing EHR systems, organisational goals may be the only requirements that are stated explicitly, as these are essential for making the political case for the systems. For example, with respect to England's NPfIT, the NHS Connecting for Health web site [2] says: *"The NHS Care Records Service will make caring for you across organisational boundaries safer and more efficient. It will also give you access to your record that covers your care across different organisations, such as the GP practice and the hospital. The purpose of NHS CRS is to allow information about you to be accessed more quickly, and gradually to phase out paper and film records which can be more difficult to access."*

Such goals and objectives are generally easily understood and accepted by the public and other stakeholders. However, there is a disconnect between these objectives and goals and what is delivered by feature modelling approaches like FODA: it is difficult to demonstrate how the features in the feature model satisfy organisational business goals. As a result, it can be difficult to demonstrate the system to be developed at early stage to major stakeholders.

There approaches in the field of requirements engineering, such as goal-driven and scenario-based requirement engineering, which may be helpful to integrate with approaches such as FODA, in order to show connections between organisational and business goals and features. This is particularly important for EHR systems, which may be controversial in the minds of certain stakeholders, where concrete arguments and evidence as to how goals are to be achieved is important to capture.

[2] `http://www.nhscarerecords.nhs.uk/what-is-the-nhs-crs`

6. CONCLUSIONS

In this paper, we briefly introduced EHR systems in England, under the auspices of the NPfIT. Healthcare information systems are amongst the most complex IT system in the world. We took a domain analysis approach to analyse the commonalities and variations of EHR systems, with the ultimate aim of identifying ways in which to improve existing implementations. In the process, we reported on challenges associated with using domain analysis, particularly FODA, for such systems. We are continuing our work in describing a prototypical architecture for EHR systems in England and are using this to propose improvements to existing architectures. At the same time, we are investigating alternative EHR systems, particularly those that take a *transformative* approach to EHR (i.e., where multiple different types of EHR are used, rather than standardising on a single type of record), and the impact that this approach has on both the FODA approach and the architecture that can be derived.

7. REFERENCES

[1] The national programme for IT in the NHS: progress since 2006. Public Accounts Committee. `http://www.publications.parliament.uk/pa/cm200809/cmselect/cmpubacc/153/153.pdf`, January 2009.

[2] I. 20514:2005. Health informatics - electronic health record - definition, scope and context. Technical report, International Organization for Standardization (ISO), 2005.

[3] J. Bosch. *Design and Use of Software Architectures: Adopting and Evolving a Product-line Approach*. Addison-Wesley, 2000.

[4] H. Chen, H. Atabakhsh, S. Kaza, B. Marshall, J. Xu, G. Wang, T. Petersen, and C. Violette. Bordersafe: cross-jurisdictional information sharing, analysis, and visualization. In *dg.o 2005: Proceedings of the 2005 national conference on Digital government research*, pages 241–242. Digital Government Society of North America, 2005.

[5] S. Clamp and J. Keen. The value of electronic health records: A literature review. Technical report, Yorkshire Centre for Health Informatics, University of Leeds, 2005.

[6] C. Clancy. Key capabilities of an electronic health record system: Letter report. Technical report, Institute of Medicine, 2003.

[7] P. Clements and L. Northrop. *Software Product Lines: Practices and Patterns*. Addison-Wesley Longman, 2001.

[8] M. Corporation. Electronic health records overview. Technical report, National Institues of Health, National Center for Research Resources, April 2006.

[9] P. Donohoe, editor. *Software Product Lines: Experience and Research Directions*. Kluwer Academic Publishers, 2000.

[10] P. Dyke. Healthy connections? *Public Finance*, September:24–26, 2003.

[11] D. Fey, R. Fajta, and A. Boros. Feature modeling: a meta-model to enhance usability and usefulness. In G. Chastek, editor, *Proceedings of the Second Software*

Product Line Conference, pages 198–216. Springer, 2002.

[12] G. E. T. Force. Electronic health records: A global perspective. Technical report, HIMSS Enterprise Systems Steering Committee, 2008.

[13] M. Herbert. Professional and organizational impact of using patient care information systems. *Medinfo*, 9:849–853, 1998.

[14] K. Herbst, P. Littlejohns, J. Rawlinson, M. Collinson, and J. Wyatt. Evaluating computerised health information systems: Hardware, software and human ware: Experiences form the northern province, south africa. *J Public Health Med*, 21:305–310, 1992.

[15] K. Kang, J. Lee, and P. Donohoe. Feature-oriented product line engineering. *IEEE Software*, 9(4):58–65, 2002.

[16] K. Kang, K. Lee, and J. Lee. *Domain Oriented Systems Development:: Practices and Perspectives*, chapter Feature Oriented Product Line Engineering: Principles and Guidelines, pages 19–36. C&C, 2002.

[17] K. C. Kang, S. G. Cohen, J. A. Hess, W. E. Novak, and A. S. Person. Feature-oriented domain analysis (FODA) feasibility study. Technical Report CMU/SEI-90-TR-21, Software Engineering Institute (SEI), Carnegie Mellon University, 1990.

[18] L. Sato. A NHS logical health record architecture: Vision, objectives and success criteria. Technical report, National Program for IT, NPFIT-FNT-TO-SCG-0019.04, 2008.

APPENDIX

The extra features (derived from [18]) for the future needs of England's EHR programme are summarised in the following list.

F1 Storing the clinical data.

1. Capturing data that fulfils medico-legal requirements for patient records.
2. Capturing data that will be re-used for high-quality clinical communications within and between organisations.
3. Storing data for clinical research or central returns where appropriate.

F2 Processing the data.

1. Supporting Aggregation, integration and/or selectively viewing of current and historical patient data from a variety of sources, in a timely, reliable, safe and meaningful way.

F3 Supporting decisions.

1. Assisting with decision support that is based on the particular requirements of an individual patient.

F4 Administrative management.

1. Auditing practices and their service or service provider with a view to provide comparisons with peers using data that is clinically meaningful, and that supports the objective of improving care outcomes.
   (a) Recording patient data that can be compared against pathway expectations.
   (b) Recording patient data that can be compared against 'markers' for desired (or undesired) clinical outcomes.

F5 Supporting Patient involvement.

1. Using applications to personalise pathways or journeys that meet the individual needs of a presenting patient.
2. Supporting automated personal pathways, which will help generate information to share with the patient.
3. Storing the history appropriately for the patient's care.
   (a) Sharing the historical data across clinicians and other carers.

How to Compare Program Comprehension in FOSD Empirically – An Experience Report

Janet Feigenspan
Metop Research Center
Magdeburg, Germany
janet.feigenspan@metop.de

Christian Kästner
University of Magdeburg
Magdeburg, Germany
ckaestne@ovgu.de

Sven Apel
University of Passau
Passau, Germany
apel@uni-passau.de

Thomas Leich
Metop Research Center
Magdeburg, Germany
thomas.leich@metop.de

ABSTRACT

There are many different implementation approaches to realize the vision of feature-oriented software development, ranging from simple preprocessors, over feature-oriented programming, to sophisticated aspect-oriented mechanisms. Their impact on readability and maintainability (or program comprehension in general) has caused a debate among researchers, but sound empirical results are missing. We report experience from our endeavor to conduct experiments to measure the influence of different implementation mechanisms on program comprehension. We describe how to design such experiments and report from possibilities and pitfalls we encountered. Finally, we present some early results of our first experiment on comparing the CPP tool with the CIDE tool.

Categories and Subject Descriptors: D.2.2 [**Software**]: Software Engineering—*Design Tools and Techniques*; D.3.3 [**Software**]: Programming Languages—*Language Constructs and Features*

General Terms: Experimentation, Human Factors, Languages

Keywords: Program Comprehension, Empirical Software Engineering, FOSD, Preprocessors, CIDE

1. INTRODUCTION

In software development, a large amount of money is spent on software maintenance [29]. One major part of maintaining software is understanding code [42]. Therefore, one important goal in software engineering is to develop concepts, languages, and tools that aid understanding in order to reduce maintenance costs.

One paradigm that aims at increasing understandability is *feature-oriented software development (FOSD)* [4]. The key abstraction of FOSD is a *feature*, which represents a product characteristic or domain abstraction relevant to stakeholders. FOSD aims at separation of concerns in terms of features, even for crosscutting and interacting features, and provides corresponding abstraction and implementation mechanisms [4, 8, 36]. Modularizing software in terms of features promises improved understandability, because concerns can be traced directly from the problem space (domain description) to the solution space (implementation) [4, 30].

There are numerous implementation approaches for FOSD. Examples are preprocessor-based implementations with the C preprocessor [20], XVCL [21], or CIDE [24]; *aspect-oriented programming (AOP)* [26] with languages such as AspectJ [27]; and *feature-oriented programming (FOP)* [36] with languages or tools such as Jak/AHEAD [8] or FeatureC++ [3]. Although all approaches aim at the common goal of separation of concerns, they use very different mechanisms, ranging from annotations with `#ifdef` directives, over superimposition, to sophisticated weaving mechanisms. Which of the FOSD approaches has the best effect on understandability? Due to the immense differences between all these approaches, this cannot be answered easily. If we asked developers, we would get very different opinions [5]. Is there a way to evaluate understandability of different FOSD approaches in a sound way that is not just based on subjective opinions?

Recently, we naively set out to conduct an experiment to compare FOSD approaches empirically. Initially, we wanted to measure understandability, from here on referred to as *program comprehension*, for all common FOSD approaches and provide a ranking or results like "developers are able to understand an AspectJ-based implementation by 35 % faster than an equivalent preprocessor-based implementation" [5]. We soon realized that this would not be so simple: (a) comparing complete FOSD approaches to derive a ranking, e.g., AOP vs. FOP, is nearly impossible, because of the number of parameters that would have to be considered, and thus (b) only few aspects of program comprehension can be measured feasibly.

In this paper, we report from our experience of designing such experiment and from possibilities and pitfalls we encountered. We present some early results of our first experiment on comparing CPP with CIDE. This way, we want to convey some intuition on the boundaries of measuring program comprehension and show what can be evaluated empirically and what requires an unrealistically high amount of effort. We hope that other researchers pick up empirical evaluations, so that we can eventually combine the results to a larger picture on program comprehension in FOSD approaches.

2. BACKGROUND

In this section, we give a brief overview of different implementation mechanisms. We selected four mechanisms: (1) aspect-oriented programming with AspectJ, (2) feature-oriented program-

FOSD'09, October 6, 2009, Denver, Colorado, USA.

```
1 public class Stack {
2   LinkedList items =  new LinkedList();
3   public void push(Object i) { items.addFirst(i); }
4   public Object pop () { return items.removeFirst(); }
5 }
```

Figure 1: Base implementation of a stack.

```
1 public aspect Safe {
2   pointcut safePop(Stack stack):
3     execution(Object pop()) && this(stack);
4   Object around(Stack stack): safePop(stack) {
5     if (stack.items.size() > 0) return proceed();
6     return null;
7   }
8 }
```

Figure 2: Aspect-oriented implementation of *Safe*.

ming with Jak, (3) annotating feature code with CPP's `#ifdef` directive, and (4) annotating features in CIDE. We selected AspectJ, Jak, and CPP because their influence of program comprehension is controversially discussed [5, 13, 30, 31, 41]. CIDE was selected because it is part of our own research.

We use the running example of a class *Stack* in Figure 1, which we subsequently extend with a feature *Safe* to ensure that no elements can be popped of an empty stack.[1]

Aspect-oriented programming with AspectJ.

AOP was developed to modularize otherwise scattered and tangled code of crosscutting concerns in aspects [26]. An aspect can alter the structure of the base program by means of inter-type declarations and can alter the behavior of the base program by means of advice, which is executed at certain join points selected by pointcuts. AspectJ is a popular aspect-oriented language based on Java that implements these concepts and provides a compiler to weave aspects into Java code [27]. Several researchers have shown that AOP is suitable to implement features, e.g., Figueiredo et al. [15].

To illustrate AOP, we show one possible AspectJ implementation of feature *Safe* in Figure 2. The extension to the base stack is encapsulated in an aspect called `Safe` that defines the pointcut `safePop`. The pointcut captures the execution of the method `pop`. Advice, declared with `around`, ensures that the method pop is executed only if the stack is not empty; otherwise, it returns `null`.

Feature-oriented programming with Jak.

FOP has a similar goal to modularize crosscutting concerns, but uses a different implementation strategy: Classes are split into class fragments according to features and all class fragments of a feature are encapsulated in a feature module [8, 36]. Each class fragment contains only the part of the class that is necessary to implement the corresponding feature. To generate a program, all class fragments of a class are composed during compilation.

In Figure 3, we show the implementation of the feature *Safe* with Jak, an FOP dialect of Java implemented as part of the AHEAD tool suite [8]. The keyword `refines` is used to indicate that the declaration specifies not a full class but only a fragment (called class refinement) and the keyword `Super` is used to specify how to methods can be merged. To compile a program that includes the feature *Safe*, the class fragments of Figures 1 and 3 are composed. During composition, members are merged into a single class and methods with identical names are merged (the original method is inlined at the *Super* call)).

[1] We omit the usual method `top` for brevity.

```
1 refines class Stack {
2   Object pop () {
3     if (items.getSize() > 0) return Super.pop();
4     return null;
5   }
6 }
```

Figure 3: Feature-oriented implementation of *Safe*.

```
1 public class Stack { //...
2   public Object pop() {
3 #ifdef SAFE
4     if (items.size == 0) return null;
5 #endif
6     return items.removeFirst();
7   }
8 }
```

Figure 4: CPP implementation of the stack.

Annotations with the C preprocessor (CPP).

A different way to implement features comes from textual preprocessors like CPP [20] or those used in various product line tools like XVCL [21], pure::variants[2], or Gears[3]. Instead of separating features into distinct files (physical separation of concerns), they are only annotated in a common tangled implementation. With these annotations, we can trace each feature from the problem space to its scattered implementation.

With CPP, developers use `#ifdef` and `#endif` directives to annotate code. Since the CPP is a text-based processor, it can also be applied to other programming languages than C. Furthermore, everything can be annotated, even just an opening bracket, which is very flexible but leaves creating reasonably annotated source code to the discipline of the programmer. Academics often criticize preprocessor annotations for negative effects on readability and maintainability [13, 41], but due to their simplicity they are common in industry.

In Figure 4, we show the implementation of *Safe* using CPP. Lines 7–9 assure that elements can only be popped from a non-empty stack.

Annotations with CIDE.

CIDE was developed at the University of Magdeburg [24]. Instead of textual annotations like with CPP, in CIDE, features are annotated in a tool infrastructure and are represented with background colors, one color per feature. Additionally, modularity is emulated by providing views on the source code (e.g., show only the code of feature X) [25]. A further difference to CPP is the kind of allowed annotations: While CPP works on plain text, thus allowing us to annotate everything, CIDE uses the underlying structure of the according source code file to enforce disciplined annotations, assuring that not arbitrary text, but only classes, methods, or statements can be annotated [24].

In Figure 5, we show our stack example a last time, this time with a gray background color to denote the feature *Safe*.

Which approach is the most understandable?.

So, which of the presented FOSD approaches provides most benefit on program comprehension? As the examples demonstrate, the approaches differ considerably, so this question cannot be answered easily. For example, AOP uses a sophisticated join point model, whereas FOP creates class fragments inside feature modules. CPP

[2] `http://www.pure-systems.com`
[3] `http://www.biglever.com/`

```
1 public class Stack { //...
2   public Object pop() {
3     if (elements.size == 0) return null;
4     return elements.removeFirst();
5   }
6 }
```

Figure 5: CIDE implementation of the stack.

and CIDE even use very simple mechanisms and do not separate feature code at all. The syntax of AspectJ has a large number of new concepts and keywords, whereas in Jak and CPP, two new keywords suffice. On the other hand, compared to Jak, AspectJ is more expressive to extend a base program, and CPP allows almost every kind of changes. Do these constructs provide a benefit on program comprehension or a drawback? Do circumstances exist in which AspectJ is more comprehensible than Jak and vice versa? What about CPP and CIDE? In the remainder of the paper, we show our experience in determining this empirically and present first results.

3. DESIGNING EXPERIMENTS ON PROGRAM COMPREHENSION

There is an overwhelming body of literature on how to conduct experiments in such a way that they are sound. If we are not careful how to conduct the experiments, we can easily get biased results. For example, when asking subjects (the individuals that participate in our experiment) how well they understand an implementation in each of our four languages, we may get subjective results that are heavily influenced by their background and personal opinion (some may have learned to use AspectJ, others may be experienced with C++ development and preprocessor usage). Therefore, we must be very careful how to design and conduct experiments. We started by briefly reviewing the literature on controlled experiments in general and give a very short overview of the most important concepts[4]. Readers already familiar with conducting experiments may skip this section.

An *experiment* is a systematic research study in which the investigator directly and intentionally varies one or more factors (*independent variables*) while holding everything else constant and observes the results (*dependent variables*) of the systematic variation [16]. From this definition, three criteria for experiments can be deduced. Firstly, an experiment must be designed such that other researchers can replicate it (*replication*). This is an important control technique in empirical research. Secondly, the variations of the factors must be intended by the investigator (*intention*). Random variations should be avoided because they prevent replication. Thirdly, it is important that factors can be varied (*variability*). Otherwise, an effect on the result cannot be observed depending on the variations of the factors.

The process of experimental research can be divided into five stages. In the first two stages, *objective definition* and *design*, the experiment is prepared. During *execution*, we run the experiment and collect data, which we analyze during *analysis*. Finally, we interpret our results during *interpretation*. We introduce these stages and relevant terms next [22].

3.1 Objective Definition

The first two steps when starting an experiment is to define the variables of the experiment and to specify hypotheses that should be tested. In our case, we have one independent variable: the FOSD approach. Since we want to assess the understandability of AspectJ, Jak, CPP, and CIDE, our independent variable has four *levels*. Our dependent variable is program comprehension, because we want to assess whether and how different FOSD approaches influence program comprehension.

A *hypothesis* is an educated guess about what should happen under certain circumstances [16]. One important criterion is that hypotheses are *falsifiable*, i.e., that we can reject them [34]. Hypotheses that are continually resistant to be falsified are assumed to be true, yet it is not proven that they are. The only claim we can make is that we found *no evidence to reject* our hypotheses.

Program comprehension is 'the process of understanding a program code unfamiliar to the programmer' [28]. Depending on the amount of domain knowledge, there are different models for how program code is understood. In *bottom up models*, a program is analyzed by examining statements and grouping them to chunks, which are iteratively abstracted to a high level understanding of source code. A programmer uses a bottom up approach when he has no knowledge of the program's domain (e.g., [33]). Otherwise, he can use his domain knowledge to create hypotheses about a program's purpose and verifies or rejects them by examining the code (top down or integrated models) (e.g., [9]).

So, how can we *measure program comprehension*? Several techniques and measures exist in the literature with differing reliability and effort in applying them (see [11] for a comprehensive survey). Typical techniques include maintenance tasks, mental simulation (e.g., pen-and-paper execution of the source code) and think-aloud protocols (i.e., subjects verbalize their thoughts during comprehending a program [1]). Usually, correctness, completeness, and time to solve a task are used as measure for program comprehension. Which measure to chose depends on the experiment, we will discuss our choice later.

So, how could our hypotheses look like? An example could be 'The number of errors of a maintenance task is lower for Jak than for AspectJ with bottom up program comprehension', which exactly defines the technique, measure, and program comprehension model to which our hypothesis is applicable.

Finally, it is imperative that we define our hypotheses before designing or actually executing the experiment, because decisions in subsequent stages depend on the hypotheses (e.g., the subjects we include or the analysis methods we apply) [16]. In addition, it prevents us from the bad practice of 'fishing for results' in our data and thus discovering random relationships between variables [12].

3.2 Design

The next step is to design our experiment so that we are able to evaluate our hypotheses. During this stage, internal and external validity as well as confounding variables have to be considered.

- A *confounding variable* is a parameter that influences the dependent variable besides variations of an independent variable [16]. In order to soundly measure the influence of FOSD approaches on program comprehension, we need to identify and control the influence of confounding variables. For example, programming experience could influence program comprehension more than using different FOSD approaches. If we accidentally distribute all the experienced programmers in one group and all the novices in another, this could overshadow our measures and cause biased results.
- *Internal validity* describes the degree to which the value of the dependent variable can be assigned to the manipulation of the independent variable [39]. This means that we have to control the influence of all confounding parameters (e.g., noise level or programming experience).

[4] A comprehensive discussion can be found in [14, 39].

- *External validity* is the degree to which the results gained in one experiment can be generalized to other subjects and settings [39]. The more realistic an experimental setting is, the higher is its external validity. Hence, we could conduct our experiment in a company under real working conditions with employees of the company. Now, however, our internal validity is threatened, because we cannot control the influence of confounding variables like programming experience.

When designing experiments, we have to find a compromise between both kinds of validity. For example, if we do not know how our variables interact or our resources are rather limited, we can start with experiments that maximize internal validity (e.g., by using only unpaid students of the same programming course). This is also the path we take in our experiments. Once we have established a hypothesis, we can design experiments that are more realistic.

A first step in controlling confounding variables is to identify them. Since the number of confounding parameters on program comprehension is large, we discuss them separately in Section 4. When identified, we need to control them. In literature there are a number of approaches to control confounding variables: randomized sampling, keeping the parameter constant, including parameter as independent variable, ex post analysis of the parameter, or experimental designs [2, 16]. In our experiments, we use a simple experimental design and usually randomization, assuming that statistical errors even out with a large enough sample.

3.3 Execution and Analysis

If we carefully design our experiment, running it usually the easier part. In this stage, we recruit subjects, let them complete our tasks, and collect our data as planned.

Having collected the data, we need to describe and analyze them. For describing our sample and data, we can compute some descriptive statistics, e.g., frequencies, means, or standard deviation. This information is necessary for replicating our experiment [2].

After describing the data, we can apply significance tests to evaluate our hypotheses [2]. Those tests are necessary in order to determine whether a difference we encountered is significant or just appeared randomly. Depending on the data, we can apply different tests. For example, if we want to check whether frequencies of correct answers differ between Jak and AspectJ, we use a χ^2-test [2]. If we want the check whether measured times to complete a task differ between Jak and AspectJ, we can use a t-test or Mann-Whitney-U-test, depending on how our data are distributed [2]. For analyzing two independent variables, e.g., programming experience and FOSD approach, there are further tests like ANOVA [2]. An overview of significance test and their requirements and application can be found in [2]. Statistical tools like *SPSS* or *R* help to analyze the data.[5]

4. CONFOUNDING VARIABLES ON PROGRAM COMPREHENSION

After our brief overview of controlled experiments in general, we discuss confounding variables (also called confounding parameters) on program comprehension. Identifying and controlling confounding parameters is necessary to allow us to draw sound conclusions from our result and avoid bias. During our design, we found a high number of confounding parameters (by literature review and consulting experts). Due to space limitations, we group the parameters into three categories and pick one parameter per category to explain its influence and how it can be controlled. The remaining parameters are discussed in detail in [14].

[5]see `www.spss.com` and `www.r-project.org`

Personal parameters.

Personal parameters are related to the subjects of an experiment. As example parameter, we discuss programming experience.

Consider an expert and a novice programmer, who both solve a task in an experiment. The chance that the expert programmer has dealt with a program similar to the task at hand is considerably higher than for a novice programmer. In the case an expert knows the kind of problem, but a novice does not, the cognitive processes for understanding the source code of the task are not the same. Whereas the expert *uses* his knowledge to solve a task, the novice *acquires* knowledge. Hence, to avoid biased results (we would not solely measure whether one program is more understandable than another, but additionally the effect of programming experience on program comprehension), we control the influence of programming experience in our experiment. Of the control strategies discussed in Section 3.2, we use pseudo randomization, because of the high influence of programming experience on program comprehension.

To control the influence of programming experience, we need to measure it. In the literature, programming experience is diversely understood. We found several aspects that were used, for example years of practical programming or number of programming courses at college. Since there is no common definition or questionnaire, we need to consider relevant aspects of programming experience depending on the hypotheses of an experiment. For our experiment, we used Java code and asked in how many Java projects the subjects have participated in a preliminary survey. Details of this survey can be found in [14]. Based on the measured experience, we divided the subjects into two even groups.

Besides programming experience, we identified domain knowledge, intelligence, and education as confounding parameters. Except for domain knowledge, measuring personal parameters is hard, because either no common understanding exists (programming experience, intelligence) or the measurement is difficult (intelligence, education). Depending on the hypotheses as well as human and financial resources, according means to control the influence need to be defined. We kept domain knowledge and education constant and randomized intelligence in our experiment.

Environmental parameters.

The second group of confounding parameters is specific to experimental situations that assess program comprehension as well as experiments in general. We discuss tool support in more detail.

Software development is supported by tools that foster program comprehension. However, before functionalities of a tool can be used, persons need to familiarize with it. Even after an equal amount of time, persons can use different sets of features of the same integrated development environment (IDE). Hence, letting persons use the same IDE does not control the influence of tool support sufficiently: Some subjects need to familiarize with it, whereas others may not know how to use a certain functionality. Letting subjects use their preferred IDE prevents that they have to familiarize with an unknown tool, but introduces variations in tool support.

Depending on our hypotheses, we need to control tool support: If the comprehensibility of two languages should be compared, tool support would confound the result, because not the comprehensibility of the language alone, but also how the language is supported by the tool is measured. On the other hand, if skills of subjects should be measured, letting them use their preferred tool is more advisable, because they do not have to familiarize with a new one. In our experiment, we eliminated tool support.

Other environmental parameters beyond tool support include training and motivation of subjects, noise level, position effects, ordering effects, test effects, the Hawthorne effect [37], and the

Rosenthal effect [38]. All of them can be more or less easily controlled. For example, we could pay our subjects for good performance in our experiment, *if* we have according financial resources. In our experiment, we used a warming up task to let subjects familiarize with the experimental setting and tried to keep all other parameters constant.

Task-related parameters.

Task-related parameters are caused by the experimental tasks and source code. As an example, we discuss comments.

As shown by Prechelt et al. [35], commented code is significantly easier to understand than uncommented code. Hence, for controlling the influence of comments in our experiments, comments must be comparable in different versions and must have the intended effect (e.g., support subjects in comprehension, not confusing them). We can conduct pretests, consult experts, or assess the opinion of subjects to control the influence of comments.

Further parameters include structure of source code, coding conventions, difficulty of the task, syntax highlighting and documentation. We can control the influence of most of them like the influence of comments (i.e., conduct pretests, consult experts, or assess the opinion of subjects).

5. COMPARING FOSD APPROACHES

Now, we come back to our initial goal to compare FOSD approaches. We show that, due to the sheer number of confounding parameters discussed in the previous section, the scope of experiments that soundly *and* feasibly assess the influence of FOSD on program comprehension is very small. We started by naively designing an experiment to assess the understandability of AspectJ, Jak, CPP, and CIDE in one experiment, but soon found out that we could not realistically conduct it. Then, we tried a more narrow approach by comparing AspectJ and Jak, which also turned our as unrealistic. Finally, we found that we have to proceed in small steps, as we will show.

5.1 Four Approaches

Initially, we naively wanted to compare AOP vs. FOP vs. preprocessors in general [5]. However, we found that the programming language is an important confounding parameter. Are the results the same whether we use AspectJ [27] or CaesarJ [6]? Is there a difference when we use Java, C, or some other language as host language? To explore these effects, we need to consider programming languages as independent variables as well, and there can easily be dozens of languages for AOP and FOP and even preprocessors. Optimistically assuming just five programming languages per approach, we already have to create $5 \cdot 3 = 15$ versions of a program, which all must be comparable regarding difficulty, commenting style, structuring, etc. (i.e., confounding task-related parameters). Besides creating 15 comparable programs, we must recruit subjects for 15 groups. Alternatively, we could 're-use' our subjects such that one subject works with all programming languages of one FOSD approach, but this way each subject needs considerably longer to complete our task (five programs instead of one) and we need to control several test effects.

At this point, we have not even started with other confounding parameters like programming experience (maybe Jak is easier for novices and AspectJ is faster to understand for experts) or tool support (maybe without tool support Jak is better than AspectJ but with tool support it is the other way around), and can already show that the experiment will become extremely complex and require many subjects. If we also include the effect of programming experience, tool support or other parameters, we easily reach designs where we need thousands of subjects or tasks that take several days. Hence, we cannot feasibly compare FOSD approaches as general as AOP vs. FOP in one experiment.

5.2 Two Approaches

If we restrict our comparison to AspectJ and Jak, we restrict ourselves to just two approaches with one language each. We reduce external validity, because we can only provide results about these two languages but not about AOP or FOP in general.

Still, there are many confounding variables left. In order to be able to generalize our result, we need to include several levels of programming experience, tool support, comments, etc. as independent variables. If we include only experts and novices, IDE and text editor versions, as well as commented and uncommented versions, our required number of subjects would be too large again ($2 \cdot 2 \cdot 2 \cdot 2 = 16$ groups).

Even if we fix all this, we need two comparable programs, one written in AspectJ and one in Jak. How can we make sure that they are comparable regarding their structure, comments, difficulty, etc., despite the significant differences between both languages? AspectJ and Jak differ considerably, for example, regarding the keywords, structure of source code, composition mechanism, etc. There are numerous ways of implementing the same problem in AspectJ (of course, the same is true for Jak). So, is the AspectJ program just difficult to understand due to the given implementation, or due to AspectJ's mechanisms in general? How can we eliminate the effect of different implementations of the same problem? Furthermore, we only would compare the implementation of *one* problem. Maybe for some other problems, the outcome would be reversed?

Hence, even comparing two programming languages, is nearly impossible. We can only derive results for specific implementations of specific problems. So, what aspects of an FOSD approach and its effect on program comprehension can we feasibly measure?

5.3 Realistic Comparison

In order to feasibly design experiments, we need to restrict our external validity even more. This means that we cannot test *everything* with *one* experiment in which we analyze the effect of all levels of our independent variable and all confounding parameters on program comprehension. Instead, we have to choose the scope of our experiment so small that we can reliably measure program comprehension *and* do not exceed our available resources.

Hence, a first step is to keep most confounding parameters constant. This way, our experiment has a low degree of external validity, but it can be feasibly conducted and we can draw sound conclusions. Second, we restrict our experimental design to the simplest comparison: one independent variable with two levels. This reduces the number of subjects we need. Furthermore, we can compare two levels that are rather similar. This helps us to create comparable conditions in our experiment (e.g., two versions of source code that differ only in few aspects).

Examples of feasible comparisons are the effect of a few keywords on understandability (e.g., does it make a difference to have or not have *cflow* in AspectJ, does it make a different whether Jak uses the keyword *refines* or *layer*?) or comparing programs in the same programming language, but with different annotations (CIDE vs. CPP). Furthermore, we could only use male students as subjects that have completed the same programming courses at the same university, are familiar with the same programming languages and domains and have an average IQ. In this case, we can only generalize our results to individuals with the same characteristics. The challenge is to find the right balance.

6. DEMONSTRATION EXPERIMENT

In this section, we describe an experiment comparing the effect of CPP and CIDE on program comprehension. The purpose of this description is not provide all necessary information to replicate our experiment (which is described in [14]), but to illustrate the small scope of feasible *and* sound experiments, which we derived in the previous section.

Our goal is to measure whether using colors in CIDE instead of textual annotations à la CPP has an effect on program comprehension. As shown in Section 2, both approaches are quite similar, and we expected that, when ignoring all tool support like views, the kind of annotation has no effect on program comprehension.

6.1 Objective Definition

Our independent variable has two levels: textual annotations à la CPP and annotations using background colors à la CIDE. Our dependent variable, program comprehension, was measured with four maintenance tasks (given a bug description, subjects had to find the cause in the source code and fix it). We assessed the time to solve a task and whether a task was completed successfully.

Our hypotheses are:

- There are no differences in solving time between Java-CPP-annotated and Java-CIDE-annotated source code with bottom up program comprehension.
- There are no differences in the number of completed tasks between Java-CPP-annotated and Java-CIDE-annotated source code with bottom up program comprehension.

We expect no differences, because for all tasks, subjects need to analyze source code on a textual basis. Hence, it should be irrelevant how the according source code statements are annotated.

6.2 Controlling Confounding Variables

In order to control the influence of personal parameters, we measured programming experience in a pre-test and used matching to create homogeneous groups according to programming experience (and gender). Since our sample was large enough (about 50 subjects), we assume that both groups are homogeneous according to intelligence, too.

We chose the domain of software for mobile devices, which was unfamiliar to all subjects (ensured in pre-test), thus enforcing bottom up program comprehension. Regarding education, we selected subjects that took an advanced programming course at the University of Passau, which required several basic programming courses.

In order to control environmental parameters, we conducted the experiment in a browser, not in an IDE, thus excluded an influence of tool support. We created a HTML file for every source code file. A link to every file was displayed at the left side of the screen (similar to the package explorer of Eclipse). Subjects were not allowed to use the search function of the browser.

As training, we gave one neutral introduction in one room for all subjects to CPP and CIDE with familiar source code examples. The experiment was also conducted in one room (i.e., same noise level, etc.). For controlling the influence of motivation, subjects were required to participate in our experiment to complete their course and could enter a raffle for an Amazon gift card. All subjects knew that they participated in an experiment. Due to our limited resources, we did not conduct a repeated measure (hence: excluded test effects) or switched the order of the task. To control position and ordering effect, we used a warming up task, which took about ten minutes and should subjects familiarize with the source code. Furthermore, the tasks were arranged with increasing difficulty, so that, with each task, subjects were more familiar with the source code.

In order to control task-related parameters, we used a code-reviewed Java source code for an application for mobile devices, developed by others [15]. The code has about 3800 lines of code, 37 classes, and 4 features were already annotated using textual *#ifdef* statements. For creating the CIDE version, we deleted all lines that contained preprocessor statements and colored the background of all feature source code with the same colors as CIDE uses (see [14] for details). Since we used a code-reviewed version, structure, coding conventions, comments, documentation was already approved by experts, and our changes for CIDE did not affect them. For syntax highlighting, we used the same style as Eclipse, because all of our subjects were familiar with it.

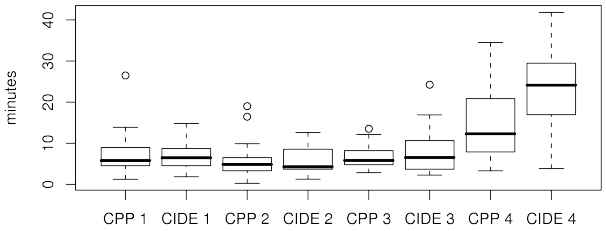


Figure 6: Response times with CPP and CIDE for all four tasks.

Since we had the same source code, we could use the same tasks for both versions. All four maintenance tasks we created by introducing bugs that occurred during runtime (forcing the subjects to examine the control flow of the program) in the source code of a specific feature. Du to space limitations we have to defer the interested reader to [14] for details on these tasks. Before the experiment, we confirmed that the bugs we produced can be found by subjects in a reasonable amount of time (within two hours) with a pre-test with some students from the University of Magdeburg.

6.3 Results

We report our results, before we interpret them. This separation is standard practice [7] to ensure that readers can distinguish results from interpretation, which helps to understand the consequences we draw from our result.

For testing our first hypothesis (no difference in solving time), we conducted a Mann-Whitney-U-test [2] to compare the mean solving time of both versions. For the first three maintenance tasks, we found no differences in solving time. For the last task, CPP subjects were significantly faster, which means that we have to reject our hypothesis. For visualization, we show box plots[6] of the times for all four maintenance tasks for each CPP and CIDE in Figure 6.

We tested our second hypothesis (no difference in number of completed task) with a χ^2 test, which checks whether observed frequencies significantly differ from expected frequencies. We found no significant differences in the number of completed task, which confirms our hypothesis.

How can those results be interpreted? Since we found a difference in response times for one task, we must reject our according hypothesis. Now, we have to interpret what this means. Why did a difference for the last task occur? The bug in the last task was located in a class that was entirely annotated with red as background color. We suspect that this color was the main reason for the performance difference in the task. To confirm this suspicion, we look through the comments subjects were encouraged to give us after

[6] A box plot is a common form to depict groups of numerical data and their dispersion. It plots the median as thick line and the quartiles as thin line, so that 50 % of all measurements are inside the box. Values that strongly deviate from the median are outliers and drawn as separate dots.

the experiment. Some subjects indeed marked this as problem and wished they could have adjusted the intensity of the background color to their needs.

Note that although we did not observe significant differences in our data (except for the last maintenance task), this does not mean that there are none. Instead, what our results show is that we have found no evidence of differences. It is possible that there are indeed effects on response time or number of correctly solved tasks, yet the effect could be too small for our sample to reveal or that other confounding variables we did not think of eliminated the effect. This is one reason why replication is crucial to empirical research: Only if a hypothesis is continually resistant to be rejected, we can assume that the relationship it describes indeed exists.

Next steps in assessing the understandability of CIDE and CPP are to replicate our experiment and confirm our second hypothesis (i.e., the kind of annotation has no effect on the number of completed tasks). Furthermore, we will explore whether the reason for the performance difference in the last maintenance task was influenced by the background color or something else.

To summarize, we learned from this experiment that, yes, it is possible to measure program comprehension given a sufficiently small scope. This encouraged us to keep on evaluating program comprehension regarding further factors, e.g., tool support in IDEs or disciplined annotations. In this experiment, we found that coloring code does not significantly increase program comprehension (at least not when searching for a bug in a feature), but, on the contrary, can even hinder it. This gave us a new perspective on our tool and encouraged us to search for other visualizations or make them adjustable by the user.

7. FURTHER WORK

So, what are next steps in measuring program comprehension? If we think of the demonstration experiment, we can extend our independent variable to other programming languages or create other programs with other degrees of complexity. We can use programming experts as subjects instead of novices. However, we cannot vary all parameters at once, but have choose very few (otherwise, we would exceed our resources).

For annotations with CPP and CIDE, it was relatively easy to create comparable programs, because we ignored tool support and the underlying programming languages are identical, so that the kind of annotation is the only difference between the versions. In a next step we will evaluate tool support.

But how can we start to compare AOP and FOP, which differ so much? The answer is, that we have to start even smaller, for example, compare two programs that only differ in their extension (`refines` in Jak vs. an inter-type declaration in AspectJ). When we have collected enough data to explain small differences between Jak and AspectJ, we can incrementally increase complexity and differences of programs and integrate the knowledge into a theory of understandability of Jak and AspectJ. In order to assess the understandability of AOP vs. FOP, we have to generalize our knowledge to other programming languages.

Since the steps in comparing AOP and FOP are rather small, it may take a decade until we have a sound body of knowledge concerning program comprehension of AOP and FOP, let alone all FOSD approaches. Nobody knows whether there is still interest in FOSD in ten years or whether other programming paradigms have emerged by then. It is impossible for one research group to assess the understandability of FOSD approach in a reasonable amount of time with a reasonable amount of financial resources. Hence, with this work, we want to encourage others to take up empirical research and establish a community for measuring program comprehension of FOSD approaches. This way, we have more people working on creating a body of knowledge, which speeds up the process and reduces the errors we make along the process.

8. RELATED WORK

We are not aware of empirical research on program comprehension in the context of FOSD. However, empirical results can be found in other domains, from which we can learn. For example, with the development of the object-oriented paradigm, researchers were curious about the benefit of object orientation compared to procedural languages. Daly et al. [10] assessed the effect of inheritance on understandability. They found performance differences in favor of object-oriented source code (although the opinion of subjects was that maintainability of procedural source code was better). A similar result was found by Henry et al. [18], who compared C and Objective C source code.

Also modeling and aspect-oriented languages have been analyzed in the past. Patig proposed a set of guidelines and a tool for testing the understandability of modeling notations [32]. This work has inspired our attempts to empirically measure program comprehension in the context of FOSD. Hanenberg et al. explored empirically whether aspect-oriented programming increases the development speed for crosscutting code [17]. They compared different kinds of tasks (different kinds of crosscutting concerns) and different kinds of languages (object-oriented and aspect-oriented).

Recently, several systematic reviews of the status of empirical software engineering were published [19,23,40]. Although they do not focus on program comprehension, they provide useful advice for empirical research in general (e.g., include experts in experiments to ensure external validity or refer to disciplines like cognitive psychology, because comparable problems occurred there along with solutions due to the age of this discipline).

9. CONCLUSION

We reported how we set out to compare FOSD approaches like AOP, FOP, or preprocessor-based implementations empirically regarding program comprehension. We learned that, in order to be able to draw sound conclusions from an experiment (internal validity), it is important to control confounding parameters on program comprehension. However, their sheer number makes large-scoped experiments difficult. It is practically impossible to compare all FOSD approaches at once. Instead, a hypothesis should focus on few aspects of FOSD approaches, because this allows us to feasibly test it. With a small experiment comparing different forms of preprocessors, we demonstrated the feasibility of small-scale experiments.

The next steps in assessing the understandability of FOSD approaches are to define small hypothesis and evaluate them empirically. Once results for those small hypotheses are clear, we can work on more complex hypotheses. In order to speed up this tedious process, it is necessary to establish a research community for empirically assessing the understandability of FOSD approaches. With our work, we hope to motivate some researches to join us.

Acknowledgments. We thank Jörg Liebig for his help on organizing the experiment in Passau. We thank METOP GmbH for the Amazon gift card for our subjects. Feigenspan's work is supported in part by BMBF project 01IM08003C (ViERforES). Apel's work is supported in part by DFG project #AP 206/2-1.

10. REFERENCES

[1] J. R. Anderson. *Cognitive Psychology and its Implications*. Worth Publishers, 2000.

[2] T. W. Anderson and J. D. Finn. *The New Statistical Analysis of Data*. Springer, 1996.

[3] S. Apel et al. FeatureC++: On the Symbiosis of Feature-Oriented and Aspect-Oriented Programming. In *Proc. Int'l Conf. Generative Programming and Component Engineering*, pages 125–140, 2005.

[4] S. Apel and C. Kästner. An Overview of Feature-Oriented Software Development. *Journal of Object Technology (JOT)*, 8(5):49–84, 2009.

[5] S. Apel, C. Kästner, and S. Trujillo. On the Necessity of Empirical Studies in the Assessment of Modularization Mechanisms for Crosscutting Concerns. In *Proc. Int'l Workshop on Assessment of Contemporary Modularization Techniques*, pages 1–7. 2007.

[6] I. Aracic, V. Gasiunas, M. Mezini, and K. Ostermann. An Overview of CaesarJ. *Transactions on Aspect-Oriented Software Development I*, pages 135–173, 2006.

[7] A. P. Association. *Publication Manual of the American Psychological Association*. American Psychological Association, 2001.

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

[9] R. E. Brooks. Using a Behavioral Theory of Program Comprehension in Software Engineering. In *Proc. Int'l Conf. Software Engineering*, pages 196–201. 1978.

[10] J. Daly et al. The Effect of Inheritance on the Maintainability of Object-Oriented Software: An Empirical Study. In *Proc. Int'l Conf. Software Maintenance*, pages 20–29. 1995.

[11] A. Dunsmore and M. Roper. A Comparative Evaluation of Program Comprehension Measures. Technical Report EFoCS-35-2000, University of Strathclyde, 2000.

[12] S. Easterbrook, J. Singer, M.-A. Storey, and D. Damian. *Guide to Advanced Empirical Software Engineering*, pages 285–311. Springer, 2008.

[13] J. Favre. Understanding-In-The-Large. In *Proc. Int'l Workshop on Program Comprehension*, page 29. 1997.

[14] J. Feigenspan. Empirical Comparison of FOSD Approaches Regarding Program Comprehension – A Feasibility Study. Master's thesis, University of Magdeburg, 2009.

[15] E. Figueiredo et al. Evolving Software Product Lines with Aspects: An Empirical Study on Design Stability. In *Proc. Int'l Conf. Software Engineering*, pages 261–270. 2008.

[16] C. J. Goodwin. *Research In Psychology: Methods and Design*. Wiley Publishing, Inc., 1998.

[17] S. Hanenberg, S. Kleinschmager, and M. Josupeit-Walter. Does Aspect-Oriented Programming Increase The Development Speed for Crosscutting Code? An Empirical Study. In *Proc. Int'l Symp. Empirical Software Engineering and Measurement*, 2009.

[18] S. Henry, M. Humphrey, and J. Lewis. Evaluation of the Maintainability of Object-Oriented Software. In *Proc. TENCON*, pages 404–409. 1990.

[19] A. Höfer and W. Tichy. *Empirical Software Engineering Issues. Critical Assessment and Future Directions*, chapter Status of Empirical Research in Software Engineering, pages 10–19. Springer, 2007.

[20] International Organization for Standardization. *ISO/IEC 9899-1999: Programming Languages—C*, 1999.

[21] S. Jarzabek, P. Bassett, H. Zhang, and W. Zhang. XVCL: XML-based Variant Configuration Language. In *Proc. Int'l Conf. Software Engineering*, pages 810–811. 2003.

[22] N. Juristo and A. M. Moreno. *Basics of Software Engineering Experimentation*. Kluwer, 2001.

[23] V. B. Kampenes et al. A Systematic Review of Quasi-Experiments in Software Engineering. *Information and Software Technology*, 51(1):71–82, 2009.

[24] C. Kästner, S. Apel, and M. Kuhlemann. Granularity in Software Product Lines. In *Proc. Int'l Conf. Software Engineering*, pages 311–320. 2008.

[25] C. Kästner, S. Trujillo, and S. Apel. Visualizing Software Product Line Variabilities in Source Code. In *Proc. Workshop Visualization in Software Product Line Engineering*, 2008.

[26] G. Kiczales et al. Aspect-Oriented Programming. In *Proc. Europ. Conf. Object-Oriented Programming*, pages 220–242. 1997.

[27] G. Kiczales et al. An Overview of AspectJ. In *Proc. Europ. Conf. Object-Oriented Programming*, pages 327–353. 2001.

[28] J. Koenemann and S. P. Robertson. Expert Problem Solving Strategies for Program Comprehension. In *Proc. Conf. Human Factors in Computing Systems*, pages 125–130. 1991.

[29] B. P. Lientz and E. B. Swanson. *Software Maintenance Management*. Addison-Wesley, 1980.

[30] R. Lopez-Herrejon, D. Batory, and W. Cook. Evaluating Support for Features in Advanced Modularization Technologies. In *Proc. Europ. Conf. Object-Oriented Programming*, pages 169–194. 2005.

[31] M. Mezini and K. Ostermann. Variability Management with Feature-Oriented Programming and Aspects. In *Proc. Int'l Symp. Foundations of Software Engineering*, pages 127–136. 2004.

[32] S. Patig. A Practical Guide to Testing the Understandability of Notations. In *Proc. Asia-Pacific Conf. Conceptual Modelling*, pages 49–58, 2008.

[33] N. Pennington. Stimulus Structures and Mental Representations in Expert Comprehension of Computer Programs. *Cognitive Psychologys*, 19(3):295–341, 1987.

[34] K. Popper. *The Logic of Scientic Discovery*. Routledge, 1959.

[35] L. Prechelt et al. Two Controlled Experiments Assessing the Usefulness of Design Pattern Documentation in Program Maintenance. *IEEE Trans. Softw. Eng.*, 28(6):595–606, 2002.

[36] C. Prehofer. Feature-Oriented Programming: A Fresh Look at Objects. In *Proc. Europ. Conf. Object-Oriented Programming*, pages 419–443. 1997.

[37] F. J. Roethlisberger. *Management and the Worker*. Harvard University Press, 1939.

[38] R. Rosenthal and L. Jacobson. Teachers' Expectancies: Determinants of Pupils' IQ Gains. *Psychological Reports*, 19(1):115–118, 1966.

[39] W. R. Shadish, T. D. Cook, and D. T. Campbell. *Experimental and Quasi-Experimental Designs for Generalized Causal Inference*. Houghton Mifflin, 2002.

[40] D. I. K. Sjoberg et al. A Survey of Controlled Experiments in Software Engineering. *IEEE Trans. Softw. Eng.*, 31(9):733–753, 2005.

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

[42] A. von Mayrhauser and A. M. Vans. Program Comprehension During Software Maintenance and Evolution. *Computer*, 28(8):44–55, 1995.

RobbyDBMS – A Case Study on Hardware/Software Product Line Engineering

Jörg Liebig and Sven Apel and Christian Lengauer
Department of Informatics and Mathematics
University of Passau, Germany
{joliebig,apel,lengauer}@fim.uni-passau.de

Thomas Leich
Metop Research Center
Magdeburg, Germany
thomas.leich@metop.de

ABSTRACT

The development of a highly configurable data management system is a challenging task, especially if it is to be implemented on an embedded system that provides limited resources. We present a case study of such a data management system, called RobbyDBMS, and give it a feature-oriented design. In our case study, we evaluate the system's efficiency and variability. We pay particular attention to the interaction between the features of the data management system and the components of the underlying embedded platform. We also propose an integrated development process covering both hardware and software.

Categories and Subject Descriptors

D.2.10 [**Software**]: Design—*Methodologies*;
D.2.11 [**Software**]: Software Engineering—*Domain-specific architectures*

General Terms

Design

Keywords

Hardware Product Lines, Software Product Lines, Domain Engineering, Feature Oriented Software Development, FeatureC++

1. INTRODUCTION

Current statistics reveal that 98 % of all microprocessors sold worldwide are part of embedded systems [18, 12]. Embedded systems play an important role in domains such as automotive, industrial automation, and control systems. Carrying out tasks in these domains involves the gathering, processing, and storage of data. Embedded systems handle data collected from several sources such as sensor data, technical service specifications, configuration, or protocol data. Although the amount of data is usually small, an efficient data management plays a crucial role, since most embedded systems come with very low computational power and only a small amount of memory.

The need for an efficient data management system in the embedded systems domain has been observed before. Researchers and engineers proposed various systems, such as IBM DB2 Everyplace [10], LGeDBMS [11], Smart Card DBMS [1], TinyDB [14], and Stones DB [8]. Each system has been developed from scratch to serve a specific application that the developers had in mind. To this end, a fixed set of data management functionalities is sufficient and, therefore, each system contains no or few variabilities. This results in a specialized software and is the reason why so many different systems have been proposed. Moreover, each of these systems is designed for a specific embedded platform, and, to this end, existing hardware variabilities are not taken into account.

A way to overcome such limited hardware and software variability is *product line engineering* (*PLE*). PLE has already been applied successfully in hardware engineering such as of cars or mobile phones, and is gaining increasing attention in software engineering [6, 15]. Since more and more products consist of hardware and software variants, the combined PLE application for hardware and software engineering is promising. However, very little is known about the impact that PLE has on products that consist of hardware and software.

To investigate the impact of PLE, we use *feature-oriented programming* (*FOP*), one implementation approach for PLE, and develop the case study RobbyDBMS: an efficient data management system for various embedded systems. We evaluate RobbyDBMS regarding efficiency and variability. Based on the results obtained, we discuss the benefits of the PLE approach. PLE has been applied successfully earlier for the development of efficient data management systems [17].[1] Although Fame-DBMS aimed at a database family for embedded systems, it did not meet our hardware requirements in terms of computational power and size of available memory.

During the implementation of RobbyDBMS, we observed that the data management system and the underlying embedded platform exhibit functional interactions. For example, for permanent data storage, a non-volatile memory or storage facility is necessary. We explain these interactions and propose an integrated development process for hardware/software PLE. Additionally, we highlight the benefits of our approach.

Specifically, we make the following contributions:

- We present our case study RobbyDBMS: an efficient data management system for embedded systems.
- We evaluate and discuss FOP as one implementation

[1] Project Fame-DBMS – `http://fame-dbms.org/`

FOSD'09, October 6, 2009, Denver, Colorado, USA.

technique for PLE. The evaluation covers the efficiency and the variability of the feature-oriented design.
- We highlight interactions between RobbyDBMS and the underlying embedded platform.
- We propose and discuss the feasibility of an integrated development process for systems like RobbyDBMS based on domain/application engineering exhibiting both hardware and software variability.

2. BACKGROUND

Product Line Engineering

Product line engineering is a concept comprising methods, tools, and techniques for the development of product lines [2]. A *product line* is a set of mutually related products that are tailored to a specific domain or market segment and that share a common set of features [9]. A *feature* represents a commonality or a variability among the set of products [19]. By selecting varying sets of features, different products (a.k.a. *variants*), fulfilling the requirements of a specific application, can be generated. Product lines are being developed in both hardware and software engineering, and we refer to them as *hardware product lines* (*HPL*) and *software product lines* (*SPL*), resp. Furthermore, we refer to features of the HPL and the SPL as *hardware features* resp. *software features*.

The process behind PLE is domain and application engineering (Figure 4) [7]. It comprises the development of reusable features in domain engineering that are used in application engineering to derive a specific product. Domain and application engineering both consist of three phases (domain engineering: domain analysis, domain modeling, and domain implementation; application engineering: requirements analysis, design analysis, and integration/test).

Feature-Oriented Programming

One approach to the implementation of SPLs is *feature-oriented programming*, which aims at the modularization of software systems using features [16, 5]. The idea is to modularize features in *feature modules* and, consequently, to obtain a 1-to-1 mapping between features as a domain abstraction and feature modules as an implementation abstraction.

In particular, we use FeatureC++ [4, 3], an extension of the programming language C++, which enables programmers to encapsulate the functionality of a feature in a feature module. A feature module represents program functionality and is composed with the base system (*feature composition*) via declarative expressions. With FeatureC++, one can add variables and methods to existing classes or one can even add new classes. Furthermore, FeatureC++ provides capabilities for extending existing methods using method refinement [4].

3. CASE STUDY ROBBYDBMS

In this section, we describe our case study of RobbyDBMS. Furthermore, we evaluate RobbyDBMS regarding efficiency and variability, and discuss the benefits of using FOP for the implementation of RobbyDBMS.

3.1 Overview

AVR HPL

The AVR HPL is a product line of 8-bit microprocessors used in the embedded systems domain. Using the AVR

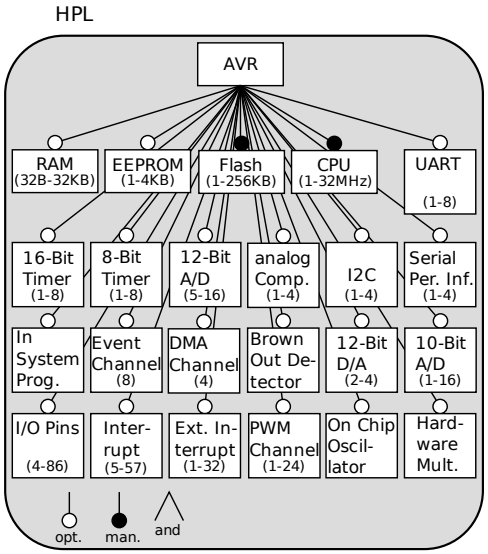


Figure 1: AVR HPL; feature diagram

HPL, programmers face a number of resource constraints, such as the low performance of the CPU (1 - 32 MHz), the small supply of program storage (1 - 256 KB), and the small main memory (32 B - 32 KB). Furthermore, different variants of the AVR series have up to 4 KB of EEPROM storage, which can be used as a permanent storage for data. These resource constraints result in cost and energy savings. AVR microprocessors are integrated in embedded systems, which have additional hardware features like sensors or actuators. Figure 1 contains a *feature diagram*, which is an excerpt of the AVR HPL *feature model* [7], representing hierarchical relationships of features in the AVR HPL. For example, feature RAM is optional and, in case this feature is available, different variants can have memory from 32 B to 32 KB.

The resource constraints mentioned before oblige to create a data management systems that is both: (1) highly configurable in terms of the features that the data management provides and (2) tailorable to a specific variant of the HPL that is being selected.

RobbyDBMS SPL

RobbyDBMS is an embedded data management system aiming at high configurability and the support of low-performance embedded systems. To increase configurability, we model most features, which are usually an integral part of data management systems, as optional (e.g., INDEXING, CHECKSUMS, BUFFERING, and TRANSACTION). This reduces the amount of program storage and main memory so as to leave as much of the system resources as possible to the programmer who makes use of the data management system. Figure 2 shows an excerpt of a feature diagram representing the feature model for RobbyDBMS.

Overall, RobbyDBMS consists of 33 feature modules with 46 classes and 37 class refinements. These refinements involve 33 method refinements, 39 additions of a function, and 15 additions of a field. The number of feature modules exceeds the number of features because of functional dependencies between different features that necessitated the split of a

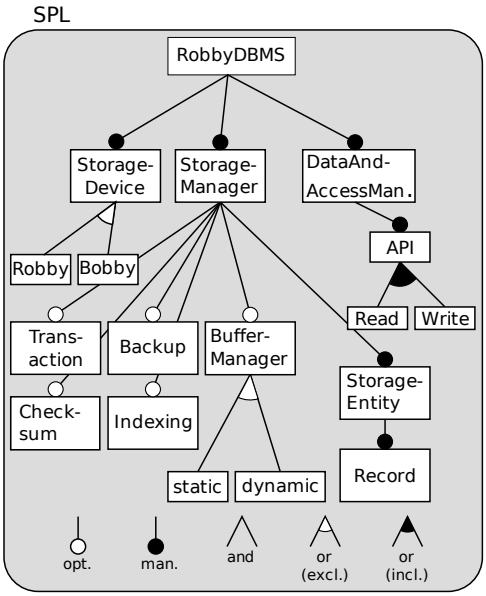


Figure 2: RobbyDBMS SPL; feature diagram

module. One example is feature CHECKSUMS that relies on READ and WRITE. Since feature WRITE is optional, we had to split feature CHECKSUMS for both variants (with and without write support).

3.2 Evaluation

The usage of FeatureC++ for developing SPLs has been evaluated before [13, 17]. Our results coincide with these evaluations and exhibit further insights on the efficiency and variability.

Efficiency

To evaluate the efficiency of FeatureC++, we developed several test programs that record the overhead in program storage (program size) and main memory caused by the use of FeatureC++. Table 1 displays the results of our analysis. The numbers reveal that, starting from a minimal variant of RobbyDBMS (only feature READ is selected), different variants with very little consumption of program storage and main memory can be generated. Furthermore, the table shows that the consumption of program storage and main memory increases almost linearly with the number of features selected. The overall deviation between the increase in program size caused by each feature separately and by the composition of all features is less than 2 %. To this end, the composition mechanisms of FeatureC++ do not introduce an overhead. Beside the generation of RobbyDBMS variants and the measurement of their memory footprints, we also investigated two optimization strategies for the composition.

First, we investigated whether the order of features has an impact on the program size. We expected that the order influences the program size, since a different order may activate different optimizations of the compiler during program compilation. To this end, we selected two features (CHECKSUMS and INDEX), that are independent but that address partially the same classes, and measured the memory footprint. We observed that the order of the two features has a small, but measurable impact ($\sim$ 0,53 %), on the program size. A deeper analysis of the savings revealed that the resorting affected multiple places in the binary and a clear relation to the resorting was not possible. Thus, we were not able to draw any conclusions; the savings might have occurred only by chance. However, we suspect that the order of features has only a very little effect on the program size. Usually, a C++ compiler applies several optimizations during the program compilation. These optimizations also involve the resorting of source code pieces, which exceeds FeatureC++'s resorting capabilities.

variant	Read	Write	Buffering	Transaction	Checksums	Index	program size	main memory
1	✓	o	o	o	o	o	658	0
2	✓	✓	o	o	o	o	996	0
3	✓	✓	o	o	✓	o	1278	0
4	✓	✓	o	o	o	✓	1686	0
5	✓	✓	o	o	✓	✓	2002	0
6	✓	✓	✓	o	o	o	2294	10
7	✓	✓	✓	o	✓	✓	3330	10
8	✓	✓	✓	✓	✓	✓	3734	12

✓feature selected; o feature not selected

Table 1: Different RobbyDBMS variants with the used program storage and main memory (in bytes)

Second, we investigated whether the program size could benefit from the application of optimization directives. For example, an optimization directive forces a compiler to apply methods for reducing the program size. FeatureC++'s compostion mechanisms rely on function inlining, which can be controlled by such directives.[2] We observed that, when several features refine a function, function inlining is not applied automatically by the C++ compiler. However, inlining can be enforced by the optimization directive `__attribute__((always_inline))`.[3] We measured that the application of this directive reduces the program size by 6.3 %. Although the reduction seems to be small, a different variant of the underlying HPL may be applicable. Furthermore, the C++ compiler used provides additional optimization directives that may further reduce the program size.

Our analysis reveals that FeatureC++ is appropriate in our case study. However, further research is necessary to show that our observations also hold in different domains and case studies.

Variability

Using PLE, we were able to model and implement a data management system that can be configured to a large extent based on features, such as BUFFERING, INDEXING, or TRANSACTION. The overall number of features is 19: 7 mandatory features that subdivide the base system and 12 optional

[2]Function inlining instructs the compiler to replace a function call with the body of the called function.

[3]This optimization directive is limited to the GCC compiler (`http://gcc.gnu.org`). However, other compilers have similar method modifiers, such as `__forceinline` in Visual C++.

features, which can be added via feature selection. However, a software feature like TRANSACTION represents only the variability of the SPL in terms of data management functionality.

The tailoring of the data management to a specific embedded systems platform is handled in the RobbyDBMS SPL, too. This includes the development and use of device drivers. Since hardware features are also variable (Figure 1; the EEPROM storage ranges from 1 KB to 4 KB) we used PLE for the development of these device drivers.

While implementing RobbyDBMS, we observed that a software feature requires one ore more hardware features. This requirement arises from functional interactions between hardware and software features. In the next section, we address this issue and discuss the influence of interactions between hardware and software features on the PLE process.

4. INTERACTIONS BETWEEN THE HPL AND THE SPL

In Figure 3, we give an example of possible interactions between the HPL and the SPL. Hardware feature EEPROM interacts with software features READ and WRITE, which can be traced back to the activity of the EEPROM as a data storage. Furthermore, feature BACKUP, which triggers write-backs to the EEPROM in case the data is held partially in main memory, either relies on feature 16-BIT TIMER or feature 8-BIT TIMER. Each hardware feature, such as EEPROM or 8-BIT TIMER, can be used by software features. We denote interactions between features on either side of the figure with bidirectional, dashed arrows. Although the necessity for a hardware feature arises on the SPL side, we show next that the interaction actually goes both ways and both sides influence each other.

Interactions between hardware and software features are bidirectional. As stated before, a software feature interacts functionally with hardware features. For example, software feature BACKUP relies on two different hardware features. From the product developer's point of view, the opposite direction, i.e., the dependence from the HPL to the SPL, becomes useful. This way, a hardware feature determines all possible variants of the SPL, which can be deployed on the HPL variant chosen. For example, hardware feature 8-BIT implies that 8 different variants of the SPL can be generated. The interactions observed allow to create a restricted feature model for the HPL and the SPL.

The treatment of interactions between HPLs and SPLs complicates the development of product lines. We believe that an integrated analysis and development process is necessary to face this complexity. In the following, we propose and discuss such an integrated process and highlight its benefits.

HPL and SPL Codesign.

In Figure 4, we depict our proposal of an integrated development process covering both hardware and software. It is based on domain engineering, where reusable features are being developed, and application engineering, where the reusable features developed are being used to generate a specific product. The figure illustrates that both domain and application engineering constitute chains of processes, such as analysis, design, and implementation, and that both chains are linked. We propose to extend this process with another ingredient covering the development of the HPL. While the

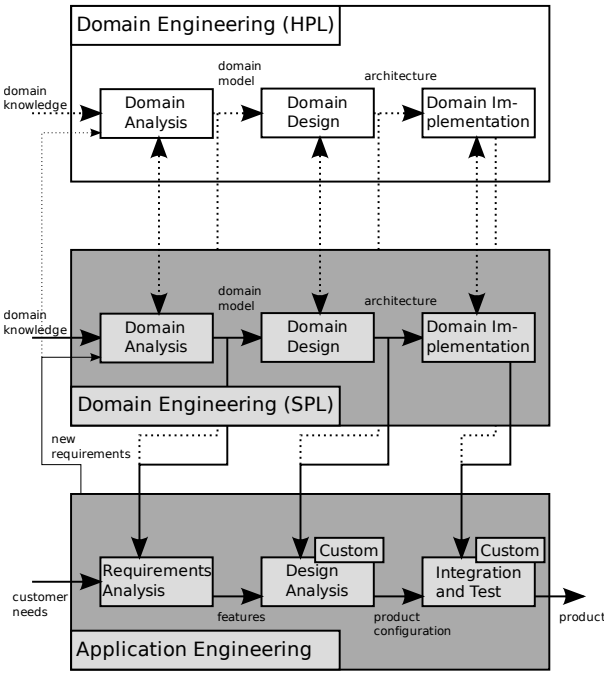


Figure 4: Integrated HPL and SPL development

output of each phase of HPL domain engineering is linked to its corresponding phase in application engineering, all phases of both domain engineering processes are linked mutually as well.

The links between the domain engineering phases cover the results of each phase (domain model, architecture, and implementation). We have already highlighted the interaction between domain models, looking at feature models, which constitute one form of representation of domain models (Figure 3). An interaction between both domain design phases covers architectural design decisions based on patterns used to describe a generic structure to achieve a highly configurable system. Finally, an interaction between domain implementations can be traced back to the software development. We have highlighted this interaction, too, as hardware features and their corresponding device drivers interact with software features.

We are aware that a full hardware/software codesign is not possible when dealing with a fixed HPL that a manufacturer supplies. However, we highlighted that PLE can benefit from taking the HPL in the process of domain engineering into account. The benefit arises from choosing among variants of an HPL.

5. PERSPECTIVE

In future work, we will investigate whether the observations we made and our proposed concept of an integrated development process also apply to other domains like operating systems for embedded system platforms. To this end, we plan to conduct further case studies. We will also study domains other than embedded systems. An interesting case study might be the Linux kernel, which consists of a huge number of features, which also relies on many capabilities

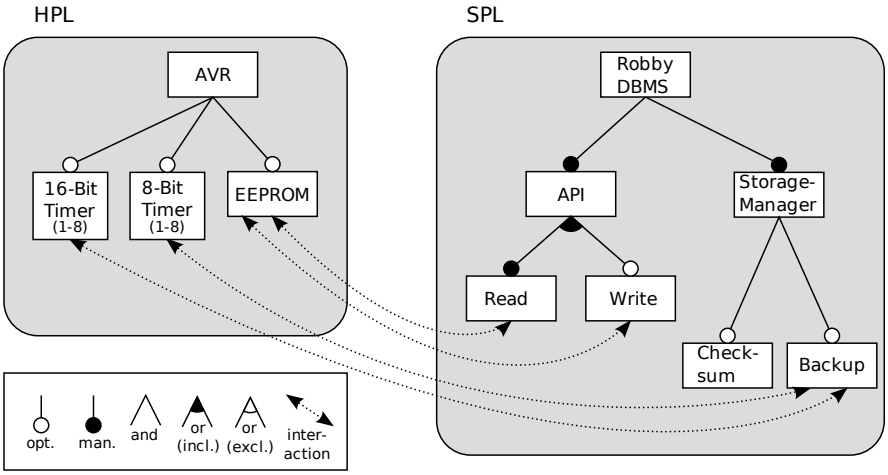


Figure 3: Example interactions between the HPL and the SPL

that different supported hardware platforms provide.

Besides conducting further case studies, we will also study each link between both domain engineering phases in more detail. The HPL used for this case study comprises a defined set of variants. An interaction between an HPL and an SPL might even be stronger than the ones we have observed so far. One reason for a stronger interaction is the existence of two technologies: (1) FPGAs[4] and (2) hardware description languages, such as VHDL[5] or Verilog. An FPGA is a computer chip, which contains programmable logic components that can be configured by customers. FPGAs are being programmed in hardware description languages. This way, changing requirements of customers regarding the hardware can also be handled. Furthermore, FPGAs add a further degree of freedom to PLE, since a feature, such as decoding a data stream, can be realized either in hardware with programmable logic or in software with a program running on a general purpose computer. Using both technologies together, the border between HPLs and SPLs becomes blurred and we expect that domain engineering for HPLs and SPLs consolidates.

6. CONCLUSION

We have reported on the development of an efficient data management system, RobbyDBMS, using product line engineering and employing the paradigm of feature orientation. We found that a feature-oriented design is suitable for modularizing variability in software like a data management system. We have achieved great variability in terms of data management functionalities and support for different embedded platforms. While implementing RobbyDBMS, we observed that the data management system and the underlying embedded platform interact and both the hardware and the software variability have to be taken into account. Consequently, we proposed a unified development process based on application and domain engineering, which combines hardware and software variabilities enabling an easier development of product lines that consist of hardware and software.

[4]field programmable gate arrays

[5]very high speed integrated circuit hardware description language

Acknowledgments

This work is being supported in part by the German Research Foundation (DFG), project number AP 206/2-1 and by the Metop Research Center.

7. REFERENCES

[1] N. Anciaux, L. Bouganim, and P. Pucheral. Smart Card DBMS: where are we now? Technical Report 80840, Institut National de Recherche en Informatique et Automatique (INRIA), Juni 2006.

[2] S. Apel and C. Kästner. An overview of feature-oriented software development. *Journal of Object Technology (JOT)*, 8(5):49–84, 2009.

[3] S. Apel, T. Leich, M. Rosenmüller, and G. Saake. FeatureC++: Feature-Oriented and Aspect-Oriented Programming in C++. Technical Report 3, Fakultät für Informatik, Universität Magdeburg, April 2005.

[4] S. Apel, T. Leich, M. Rosenmüller, and G. Saake. FeatureC++: On the Symbiosis of Feature-Oriented and Aspect-Oriented Programming. In *Proceedings of the International Conference on Generative Programming and Component Engineering (GPCE)*, volume 3676 of *Lecture Notes in Computer Science*, pages 125–140. Springer-Verlag, 2005.

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

[6] P. Clements and L. Northrop. *Software Product Lines: Practices and Patterns*. Addison-Wesley, 2002.

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

[8] Y. Diao, D. Ganesan, G. Mathur, and P. Shenoy. Rethinking Data Management for Storage-centric Sensor Networks. In *Proceedings of the Conference on Innovative Data Systems Research (CIDR)*, pages 22–31, 2007.

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

[10] J. Karlsson, A. Lal, C. Leung, and T. Pham. IBM DB2 Everyplace: A Small Footprint Relational Database System. In *Proceedings of the International Conference on Data Engineering (ICDE)*, pages 230–232. IEEE Computer Society, 2001.

[11] G.-J. Kim, S.-C. Baek, H.-S. Lee, H.-D. Lee, and M. Joe. LGeDBMS: a Small DBMS for Embedded System with Flash Memory. In *Proceedings of the International Conference on Very Large Data Bases (VLDB)*, pages 1255–1258. ACM Press, 2006.

[12] R. Krishnan. Future of Embedded Systems Technology. Technical Report GB-IFT016B, BCC Research, Juni 2005.

[13] M. Kuhlemann, S. Apel, and T. Leich. Streamlining Feature-Oriented Designs. In *Proceedings of International Symposium on Software Composition (SC)*, volume 4829 of *Lecture Notes in Computer Science*, pages 168–175. Springer-Verlag, 2007.

[14] S. Madden, M. Franklin, J. Hellerstein, and W. Hong. TinyDB: An Acquisitional Query Processing System for Sensor Networks. *ACM Transactions on Database Systems (TODS)*, 30(1):122–173, 2005.

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

[16] C. Prehofer. Feature-Oriented Programming: A Fresh Look at Objects. In *Proceedings of the European Conference on Object-Oriented Programming (ECOOP)*, volume 1241 of *Lecture Notes in Computer Science*, pages 419–443. Springer-Verlag, 1997.

[17] M. Rosenmüller, S. Apel, T. Leich, and G. Saake. Tailor-Made Data Management for Embedded Systems: A Case Study on Berkeley DB. *Data and Knowledge Engineering (DKE)*, 2009.

[18] D. Tennenhouse. Proactive Computing. *Communications of the ACM*, 43(5):43–50, 2000.

[19] P. Zave. An Experiment in Feature Engineering. In *Programming Methodology*, pages 353–377. Springer-Verlag, 2003.

Towards Systematic Ensuring Well-Formedness of Software Product Lines

Florian Heidenreich
Lehrstuhl Softwaretechnologie
Fakultät Informatik
Technische Universität Dresden, Germany
florian.heidenreich@tu-dresden.de

ABSTRACT

Variability modelling with feature models is one key technique for specifying the problem space of software product lines (SPLs). To allow for the automatic derivation of a concrete product based on a given variant configuration, a mapping between features in the problem space and their realisations in the solution space is required. Ensuring the correctness of all participating models of an SPL (i.e., feature models, mapping models, and solution-space models) is a crucial task to create correct products of an SPL. In this paper we discuss different possibilities for checking well-formedness of SPLs and relate them to their implementation in the FeatureMapper SPL tool.

Categories and Subject Descriptors

D.2.2 [**Software Engineering**]: Design Tools and Techniques—*Computer-aided software engineering, Object-oriented design methods*; D.2.2 [**Software Engineering**]: Software/Program Verification—*Validation*; D.2.13 [**Software Engineering**]: Reusable Software

General Terms

Design, Languages

Keywords

Software product lines, separation of concerns, variability modelling, well-formedness rules, FeatureMapper

1. INTRODUCTION

A software product line (SPL) is a set of software-intensive systems sharing a common, managed set of features that satisfy the specific needs of a particular market segment or mission and that are developed from a common set of core assets in a prescribed way [4]. In addition to the shared core assets, every system of a SPL has features that are specific to the system and that are not shared by all other systems (often called *products*) of the SPL. To express this variability, variability modelling is used to describe the different features available in an SPL and their interdependencies. In the Feature-Oriented Software Development (FOSD) community [2], a widely used approach for variability modelling are feature models [13, 5].

Feature-based variability modelling resides in the *problem space* whereas the realisation of features is part of the *solution space* [6]. To instantiate products from an SPL, feature realisations in the solution space have to be configured according to the presence of the features in a *variant model*; that is, a concrete selection of features from a feature model that describes a product of the SPL. This requires a mapping between features from a feature model and solution-space models or modelling artefacts that realise features (or combinations of those). We focus on model-driven development of SPLs in this paper and refer to solution-space models that are expressed by means of Ecore-based metamodels. To achieve the required mappings, a number of different approaches have been proposed [5, 9, 12] which allow for creating mappings between features from feature models and solution-space models.

While all of the approaches provide means for creating and maintaining required mappings, ensuring the *well-formedness* of all *input models* (i.e., feature models, mapping models, and solution-space models) and all possible *output models* (i.e., solution-space models transformed based on feature selection) is a challenging task often neglected in the aforementioned approaches. In this context, by well-formedness is meant the conformance of a given model with constraints of the underlying metamodel. Note, that well-formedness goes beyond syntactical correctness in the sense that it also takes additional constraints into account that are not directly expressed in the language's metamodel. Also, creating syntax errors in modelling languages is usually not directly possible, since modelling editors work on a different level of abstraction, where it is impossible to create model elements that do not conform to the concrete syntax of the modelling language. To give examples of models, which do not respect well-formedness rules and are, hence, invalid:

- A feature model can become invalid because of contradicting cardinalities, that is, if cardinalities of child features do not comply with the cardinality of their parent feature (e.g., an alternative feature where child features are mandatory).
- A mapping can reference invalid or non-existing model

FOSD'09, October 6, 2009, Denver, Colorado, USA.

elements, e.g., model elements that were removed or changed due to refactoring of the problem space or the solution space.

- An output model can be ill-formed due to mappings that do not take into account the constraints of the language used for modelling the solution space.

We argue, that for creating valid products of an SPL, the well-formedness of input and output models needs to be ensured in a systematic way and, possibly, (automatically) validated during modelling the SPL.

In this paper we discuss well-formedness of the different participating models in an SPL and present possibilities for ensuring the well-formedness of those models. Furthermore, we want to foster discussion of open and not yet addressed issues in well-formedness of SPLs motivating the FOSD community to address them in their research and development. During discussion of the identified possibilities for validation we refer to their realisation in the FeatureMapper [12, 21] SPL tool.

The rest of the paper is structured as follows: We introduce our tool FeatureMapper in Sect. 2 and provide necessary context. In Sect. 3 we discuss various possibilities for validating and enforcing well-formedness in SPLs and relate them to their implementation in FeatureMapper. We present open issues and possibilities for further research and development in Sect. 4 and refer to related work in the FOSD community in Sect. 5. Section 6 concludes the paper.

2. BACKGROUND

FeatureMapper [12, 10, 21] is an Eclipse-based tool that allows for mapping features from feature models to arbitrary modelling artefacts that are expressed by means of an Ecore-based language [19]. These languages include UML2 [15], domain-specific modelling languages (DSLs) defined using the Eclipse Modelling Framework (EMF) [19], and textual languages that are described using EMFText [11]. The mappings can be used to steer the product-instantiation process by allowing the automatic removal from the final product being generated of modelling artefacts that are not part of a selected variant.

An overview of defining an SPL and deriving a concrete product in FeatureMapper is shown in Fig. 1. To associate features or logical combinations of features (*feature expressions*) with modelling artefacts, the developer first selects the feature expression in the FeatureMapper and the modelling artefacts in her favourite modelling editor (e.g., TOPCASED [22]). Next, she applies the feature expression to the modelling artefacts via the FeatureMapper user interface (Step 1). During product derivation, this mapping is interpreted by a FeatureMapper transformation component. Depending on the result of evaluating the feature expression against the set of features selected in the variant (Step 2), the modelling elements are preserved or removed from the model (Step 3). Model elements that are not mapped to a specific feature expression are considered to be part of the core of the product line and are always preserved. In addition to product derivation, the mappings are used for visualisation purposes [10].

FeatureMapper uses cardinality-based feature models [7]. These models are instances of an Ecore-based feature metamodel. Features from these feature models are related to

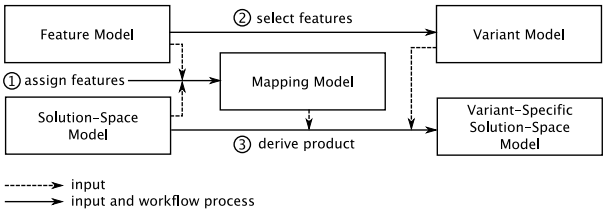


Figure 1: Workflow of defining an SPL and deriving a concrete product with FeatureMapper.

solution-space models through a dedicated Ecore-based mapping model. As depicted in Fig. 2, a mapping in this mapping model basically consists of a feature expression (`Expression`, which directly references features from the feature model or logical combinations of those) and a reference to a solution-space artefact (`EObject`).

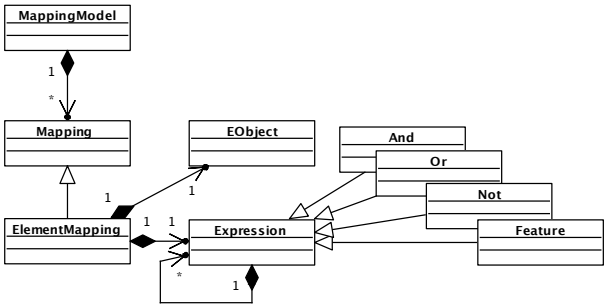


Figure 2: Simplified overview of FeatureMapper's internal mapping metamodel.

3. WELL-FORMEDNESS IN SPLS

As motivated in Sect. 1, ensuring well-formedness of all participating models is crucial in SPL to ensure the validity of the resulting products. Thus, validation of all models used for creating an SPL is needed. This includes validation of problem-space models, validation of mapping models, and validation of all possible solution-space models. In this section we give an overview of various possibilities for validation and relate them to their implementation in FeatureMapper. Our enumeration of possible validation tasks is by no means exhaustive; rather, it is expected to be extended as a result of future discussions.

3.1 Well-formed problem-space models

Cardinality-based feature models and variant models [7] include a set of well-formedness rules that need to be ensured for producing valid feature models. These rules are normally enforced by the feature-modelling tool used for producing the feature models and the variants. While FeatureMapper supports feature models and variant models of the feature-modelling tools pure::variants [3] and fmp [1], it also provides basic means to create those models, and thus, needs to ensure their validity.

FeatureMapper currently imposes the following constraints on feature models:

FM-Mandatory-Root The root feature must be mandatory. This constraint prohibits the creation of empty products by enforcing the inclusion of the root feature in any possible variant.

FM-Cardinality-Match Cardinalities of child features must comply with the cardinality of their parent feature. This constraint ensures that no contradicting cardinalities exist between child features and their parent features (e.g., an alternative feature that has mandatory child features).

FM-Sound-Reference References such as `requires` or `conflicts` must be non-contradicting. This constraint also includes the parent-child relationship between features during validation.

FM-Existing-Reference Referenced features must exist. This constraint ensures that any of the referenced features in `requires` or `conflicts` references are features in the feature model.

Variant models are seen as a subset of feature models in FeatureMapper. For ensuring valid variant models, the following constraints are enforced:

VM-Mandatory-Parent If a child feature is selected, the parent feature must be selected too.

VM-Mandatory-Child If a feature is selected, all its mandatory child features must be selected too.

VM-Alternative If an alternative feature (a parent feature with a cardinality `[0..1]`) is selected, at most one of its child features must be selected.

VM-Or If an Or feature (a parent feature with a cardinality `[n..m]`) is selected, at least `n` and at most `m` of its child features must be selected.

VM-Requires If the selection of a feature requires the selection of another feature, the latter feature must be selected too.

VM-Conflicts If the selection of a feature excludes the selection of another feature, the former feature cannot be selected.

Due to the added complexity involved when validating feature references (cf. constraints *FM-Sound-Reference*, *VM-Requires*, *VM-Conflicts* listed above), we enhanced our initial OCL-based approach for validating feature models and variant models in FeatureMapper to an Web Ontology Language (OWL) based approach similarly to what is described in [23]. This validation is exposed as a validator to the EMF Validation Framework. Additionally, FeatureMapper checks for invalid feature combinations while creating feature expressions and reports possible violations of the constraints listed above to the user.

3.2 Well-formed mapping models

Mapping models are models that relate features from feature models to their realisation in solution-space models. This mapping works directly on the referenced objects and not only on symbolic representations. Since FeatureMapper intentionally uses a generic mapping model to be independent of the modelling languages used, there exist basically two well-formedness rules that need to be ensured while creating and managing a mapping model:

MM-Existing-Feature Referenced features of a mapping must exist.

MM-Existing-ModelElement Referenced solution-space artefacts of a mapping must exist.

FeatureMapper ensures these constraints automatically during loading and saving of mapping models. If a constraint violation is detected, FeatureMapper informs the modeller and provides interactive means for correcting the model. Thus, FeatureMapper prohibits the creation of invalid mapping models. Note, that this does not imply the well-formedness of the solution-space models which will be addressed in the next subsection.

3.3 Well-formed solution-space models

The most challenging task in creating model-based SPLs is to ensure that all output models conform to the well-formedness rules of the language used for creating the solution-space models. In FeatureMapper, model elements are removed from the model during product derivation if the corresponding feature expression in the mapping does not evaluate to `true` against a given variant model. Checking the well-formedness of a output model is usually done by evaluating OCL constraints on the output model. For an SPL this would imply that any possible variant needs to be created to ensure the well-formedness of the complete SPL. This is not feasible because of the large amount of possible variants that can be created out of an SPL [16]. We focus on checking the well-formedness of an SPL, not its individual products. This includes the following constraint classes:

SM-Multiplicity Multiplicities of modelling artefacts must match the multiplicities of the respective constructs described in the metamodel of the language used for modelling the solution-space models.

SM-Typing Solution-space models must conform to the modelling language's type system.

SM-Semantics Solution-space models must conform to specific semantic constraints exposed by the used modelling languages which do not fall in any of the aforementioned classes. This also applies to domain-specific languages, which can imply constraints on solution-space models that are intrinsic to a specific domain.

To our knowledge, there currently exists no approach that allows for ensuring constraints of all three constraint classes for models created in arbitrary Ecore-based modelling languages. In [8], Czarnecki and Pietroszek presented an approach for verifying feature-based model templates against well-formedness OCL constraints. In their work, they described how the well-formedness of UML models annotated with stereotypes containing feature expressions (so-called *presence conditions*) can be ensured for all possible variants of an SPL without creating any of those variants. They described how OCL well-formedness rules can be interpreted in a way that takes the feature expressions into account and provided a set of evaluation patterns for various OCL constructs in form of propositional formulas.

While this approach and its implementation only addresses well-formedness rules of UML, well-formedness rules differ significantly depending on the modelling languages used. E.g., in UML each `DecisionNode` in an activity model must

at least have one outgoing edge whereby the guards on the outgoing edges need to be unambiguous—otherwise a race condition may occur (cf. constraint-class *SM-Semantics*). Figure 3 depicts a part of an example activity diagram from a recent case study we performed in FeatureMapper, where modelling elements are coloured according to the feature expression assigned to them. The outgoing edge coloured in red is removed whenever the respective features are not part of the given variant. The same applies to the outgoing edge coloured in blue.

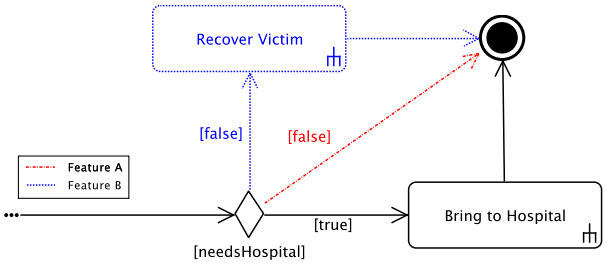


Figure 3: Example of an ambiguous activity model with two outgoing edges with the same guard condition.

Another example is, that an `Association` must have at least two `memberEnds` (cf. constraint-class *SM-Multiplicity*).[1] The detail of the class model depicted in Fig. 4 shows an `Association` where this constraint is not fulfilled in case the class `Blanket` is removed from the model. Of course, fixing those errors can be fairly easy (in this case the association needs to be removed too) but detecting those errors is a task that is not easy to perform, especially when considering cross-model constraints (e.g., each method called in a UML activity model must exist in a related UML class model; cf. constraint-class *SM-Typing*).

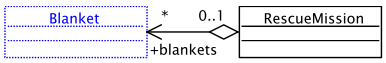


Figure 4: Example of a possible violation of the well-formedness rules of the UML `Association` concept.

Similarly to the `Association` in UML, the `eReferenceType` of an Ecore `EReference` must exist. The specifications [15, 19] contain numerous well-formedness rules based on multiplicities and OCL constraints. Presenting all of those is beyond the scope of this paper. In addition to established and widely-used modelling-languages, the trend towards defining and using DSLs in model-driven development results in numerous new languages, where each of those languages has their own set of well-formedness rules. For example, a language for creating forms can include the concept of depending questions (i.e., a question only has to be answered if a specific other question has been answered). This again involves the concept of references, where each of the referenced questions in a form description must exist.

[1]This is the running example of Czarnecki and Pietroszek in [8].

We are currently investigating possible extensions to FeatureMapper for a modelling-language independent realisation of checking the entire SPL. Since FeatuerMapper is intentionally agnostic to the modelling-languages used, any existing Ecore-based modelling-language can be used for creating solution-space models. This also means that the well-formedness rules of these different languages need to be ensured for each particular language to be supported. Our aim is at creating a generic framework that can be parameterized with those language-specific rules. To this end, ensuring well-formedness of SPLs built of arbitrary Ecore-based modelling languages becomes possible.

4. DISCUSSION

An open issue in ensuring well-formedness of SPLs is the lack of completeness of formally described well-formedness rules. The UML specification contains a lot of explicitly described multiplicities, well-formedness rules, and additional constraints but also contains implicit information (e.g., the need for unambiguousness of multiple outgoing edges on `DecisionNode`s as described in Sect. 3). To our knowledge, no catalogue of formalised descriptions (e.g., described using OCL) of those well-formedness rules exists at the moment. Even current modelling tools have a fairly relaxed interpretation of those rules and effectively allow for creating ill-formed models. Creating a complete catalogue of those well-formedness rules seems to be a complex but also very profitable task, because to this end, checking the well-formedness of all participating models of a given language is possible. To extend this idea, having such catalogues for different languages can foster reusing certain rules that are shared across languages whenever language semantics are appropriate. This is especially promising for model-driven development including multiple DSLs, where certain domain-specific constraints need to be ensured across language boundaries.

There exist a whole range of opportunities for performing additional checks on SPLs that go beyond well-formedness rules. Possible checks include detection of bad smells, i.e., violations of modelling conventions. Examples of those are direct communication between components instead of using dedicated interfaces in component models, huge inheritance hierarchies, or strong coupling of classes in class models. Possible extended checks on problem-space models can include ensuring that all features of a given feature model are actually mapped to solution-space artefacts (i.e., ensuring that there exists a realisation of a given feature in the solution-space models).

An open issue of ensuring the validity of an SPL based on different interpretation of OCL well-formedness rules is the performance of checking all propositional formulas. As described by Czarnecki and Pietroszek [8], checking those rules cannot be instantly performed due to the processing time of creating and checking those rules (in their experience, checking is performed in terms of seconds rather than milliseconds). Possible enhancements are incremental checks, where the engine detects which constraints and which parts of a model need to be verified in case of changes in the participating models. Another implication of the approach is the semantics of the mapping. It seems that this approach is feasible for mappings that relate to modelling artefacts and remove those depending on a given feature selection. Approaches that apply complex transformations based on

feature selection actually change the model in ways that are not easily verified using propositional formulas. Further research in this direction is needed.

Another widely unexplored field—which is not addressed in this paper—is detecting semantic errors for all products of an SPL. Since it is already difficult to ensure the semantic correctness of a single product, checking the semantic correctness of all possible products on an SPL is an open issue.

5. RELATED WORK

There is a whole body of work that addresses quality and safety of product lines. Some of the existing works in this field do not check the SPL itself but the distinct products that can be created out of an SPL [17]. The problem with testing all products is a large number of different products that are possible with already a fair amount of independent optional features (for n optional features, 2^n distinct variants are possible). This implies that in those cases not all possible variants are checked. Instead, only products that are actually created out of the SPL or combinatorial samples are considered which again means that a lot of repetitive inspection is needed compared to ensuring the well-formedness of the SPL itself.

As already mentioned in Sect. 3, some approaches check the SPL itself. Czarnecki and Pietroszek [8] address the problem of ensuring the validity of any possible solution-space models by checking those models against well-formedness OCL constraints. Their solution describes how well-formedness OCL constraints can be interpreted based on propositional formulas by taking into account feature expressions mapped to modelling elements. Similarly, Thaker et al. [20] use propositional formulas and SAT solvers to ensure safe composition of feature modules in the AHEAD system. In this paper we proposed extending those existing solutions to models defined in arbitrary Ecore-based languages.

In [14], Kästner et al. present an approach for guaranteeing syntactic correctness of all possible variants of an SPL. In contrast to what we discussed in this paper, their work is based on programming languages where syntax errors (such as omitting a necessary closing bracket) can easily occur. This is not the case for modelling languages since modelling editors work on a different level of abstraction where it is not possible to create model elements that do not conform to the concrete syntax of the modelling language.

In [18], Seifert and Samlaus present an approach for static analysis of source code using OCL. They present *RestrictED*, an extensible editor for textual modelling languages based on EMFText [11] that can be parametrized with language-specific constraints for checking source code modelled with EMFText languages. Our idea of creating a generic framework that can be parameterized with modelling-language specific well-formedness rules is an extension this idea, taking into account mapping information between feature models and solution-space models. Furthermore, we aim at creating a framework that abstracts from the concerte-syntax representation of the specific modelling languages (i.e., graphical or textual concrete syntax) and handles them in a uniform way.

6. CONCLUSION

In this paper we discussed various possibilities to check all participating models of an SPL against well-formedness rules defined on the metamodels that are used to create those models. We discussed existing approaches for ensuring the validity of models for all the concrete products that can be created out of an SPL as well as open issues and future work. Throughout the paper, we related the identified possibilities for ensuring well-formedness to their implementation in the FeatureMapper SPL tool and outlined our plans to integrate existing work to uniformly check well-formedness of SPLs with this tool.

Acknowledgements

We thank Ilie Şavga and Christian Wende for their valuable comments on earlier drafts of this paper. This research has been partly co-funded by the German Ministry of Education and Research (BMBF) within the project feasiPLe.

7. REFERENCES

[1] M. Antkiewicz and K. Czarnecki. FeaturePlugin: Feature Modeling Plug-In for Eclipse. In *Proceedings of the OOPSLA workshop on Eclipse technology eXchange (ETX)*, pages 67–72, New York, NY, USA, 2004. ACM.

[2] S. Apel and C. Kästner. An Overview of Feature-Oriented Software Development. *Journal of Object Technology (JOT)*, 8(5):49–84, July/August 2009.

[3] D. Beuche, H. Papajewski, and W. Schröder-Preikschat. Variability Management with Feature Models. *Science of Computer Programming*, 53(3):333–352, 2004.

[4] P. Clements and L. Northrop. *Software Product Lines: Practices and Patterns*. Addison-Wesley, 2002.

[5] K. Czarnecki and M. Antkiewicz. Mapping Features to Models: A Template Approach Based on Superimposed Variants. In *Proceedings of the 4th International Conference on Generative Programming and Component Engineering (GPCE'05)*, pages 422–437, 2005.

[6] K. Czarnecki and U. W. Eisenecker. *Generative Programming – Methods, Tools, and Applications*. Addison-Wesley, June 2000.

[7] K. Czarnecki and C. H. P. Kim. Cardinality-Based Feature Modeling and Constraints: A Progress Report. In *Proceedings of the OOPSLA'05 International Workshop on Software Factories*, 2005.

[8] K. Czarnecki and K. Pietroszek. Verifying Feature-Based Model Templates Against Well-Formedness OCL Constraints. In S. Jarzabek, D. C. Schmidt, and T. L. Veldhuizen, editors, *Proceedings of the 5th International Conference on Generative Programming and Component Engineering (GPCE'06)*, pages 211–220. ACM, 2006.

[9] Ø. Haugen, B. Møller-Pedersen, J. Oldevik, G. Olsen, and A. Svendsen. Adding Standardized Variability to Domain Specific Languages. In *Proceedings of the 12th International Software Product Line Conference (SPLC'08)*, pages 139–148. IEEE, 2008.

[10] F. Heidenreich, I. Şavga, and C. Wende. On Controlled Visualisations in Software Product Line Engineering. In *Proceedings of the 2nd International*

Workshop on Visualisation in Software Product Line Engineering (ViSPLE'08), collocated with the 12th International Software Product Line Conference (SPLC'08), Sept. 2008.

[11] F. Heidenreich, J. Johannes, S. Karol, M. Seifert, and C. Wende. Derivation and Refinement of Textual Syntax for Models. In R. F. Paige, A. Hartman, and A. Rensink, editors, *Proceedings of the 5th European Conference on Model Driven Architecture - Foundations and Applications (ECMDA-FA'09)*, volume 5562 of *LNCS*, pages 114–129. Springer, 2009.

[12] F. Heidenreich, J. Kopcsek, and C. Wende. FeatureMapper: Mapping Features to Models. In *Companion Proceedings of the 30th International Conference on Software Engineering (ICSE'08)*, pages 943–944, New York, NY, USA, May 2008. ACM.

[13] K. Kang, S. Cohen, J. Hess, W. Novak, and S. Peterson. Feature-Oriented Domain Analysis (FODA) Feasibility Study. Technical Report CMU/SEI-90-TR-0211990, Software Engineering Institute, 1990.

[14] C. Kästner, S. Apel, S. Trujillo, M. Kuhlemann, and D. Batory. Guaranteeing Syntactic Correctness for all Product Line Variants: A Language-Independent Approach. In *Proceedings of the 47th International Conference Objects, Models, Components, Patterns (TOOLS EUROPE'09)*, volume 33 of *Lecture Notes in Business Information Processing*, pages 175–194. Springer Berlin Heidelberg, June 2009.

[15] Object Management Group. UML 2.2 infrastructure specification. OMG Document, Feb. 2009. URL http://www.omg.org/spec/UML/2.2/.

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

[17] K. Pohl and A. Metzger. Software Product Line Testing. *Communications of the ACM*, 49(12):78–81, 2006.

[18] M. Seifert and R. Samlaus. Static Source Code Analysis using OCL. In J. Cabot and P. Van Gorp, editors, *Proceedings of the Workshop OCL Tools: From Implementation to Evaluation and Comparison (OCL'08), co-located with the 11th International Conference on Model Driven Engineering Languages and Systems (MoDELS'08)*.

[19] D. Steinberg, F. Budinsky, M. Paternostro, and E. Merks. *Eclipse Modeling Framework, 2nd Edition*. Pearson Education, 2008.

[20] S. Thaker, D. Batory, D. Kitchin, and W. Cook. Safe Composition of Product Lines. In *Proceedings of the 6th International Conference on Generative Programming and Component Engineering (GPCE'07)*, pages 95–104, New York, NY, USA, 2007. ACM.

[21] The FeatureMapper Project Team. FeatureMapper, July 2009. URL http://www.featuremapper.org.

[22] The Topcased Project Team. TOPCASED, July 2009. URL http://www.topcased.org.

[23] H. H. Wang, Y. F. Li, J. Sun, H. Zhang, and J. Pan. Verifying feature models using OWL. *Web Semantics*, 5(2):117–129, 2007.

An Extension for Feature Algebra

[Extended Abstract]

Peter Höfner
Institut für Informatik
Universität Augsburg
86135 Augsburg, Germany
hoefner@informatik.uni-augsburg.de

Bernhard Möller
Institut für Informatik
Universität Augsburg
86135 Augsburg, Germany
moeller@informatik.uni-augsburg.de

ABSTRACT

Feature algebra was introduced as an abstract framework for feature oriented software development. One goal is to provide a common, clearly defined basis for the key ideas of feature orientation. We first present concrete models for the original axioms of feature algebra which represent the main features of feature oriented programs. However, these models show that the axioms of the feature algebra do not reflect some aspects of feature orientation properly. Hence we modify the axioms and introduce the concept of an *extended feature algebra*. Since the extension is also a generalisation, the original algebra can be retrieved by a single additional axiom. Last but not least we introduce more operators to cover concepts like overriding in the abstract setting.

Categories and Subject Descriptors

D.2.2 [**Software Engineering**]: Design Tools and Techniques—*Object-oriented design methods*; F.3.1 [**Logics and Meanings of Programs**]: Specifying and Verifying and Reasoning about Programs—*Logics of programs,Mechanical verification,Specification techniques*

General Terms

Design, Languages, Verification

Keywords

feature oriented software development, feature algebra, algebraic characterisation of FOSD

1. INTRODUCTION

Over the last few years *Feature-Oriented Software Development* (FOSD) (e.g. [7]) has been established in computer science as a general programming paradigm that provides formalisms, methods, languages, and tools for building variable, customisable, and extensible software.

FOSD'09, October 6, 2009, Denver, Colorado, USA.

Feature algebra [3] is a formal framework that captures many of the common ideas of FOSD such as introductions, refinements, or quantification and hides differences of minor importance. It abstracts from the details of different programming languages and environments used in FOSD. Moreover, alternative design decisions in the algebra reflect variants and alternatives in concrete programming language mechanisms; for example, certain kinds of feature composition may be allowed or disallowed.

In one of the standard models of feature algebra, the structure of a feature is represented as a tree, called a *feature structure tree* (FST) [2]. An FST captures the essential, hierarchical module structure of a given program. Based on that, feature combination can be modelled as superimposition of FSTs, i.e., as recursively merging their corresponding substructures.

Feature algebra serves as a formal foundation of architectural metaprogramming [6] and automatic feature-based program synthesis [10]. Both paradigms emerged from FOSD and facilitate the treatment of programs as values manipulated by metaprograms, e.g., in order to add a feature to a program system. This requires a formal theory that precisely describes which manipulations are allowed.

In the present extended abstract we first derive a concrete model for feature algebra that is based on FSTs. It was already sketched in [3]; however we define it in a precise way. After introducing the abstract notion of a feature algebra, we present another concrete model. Next we show that the models are fine as long as one does not consider feature oriented programming on code level. If manipulation of code and not only of the overall program structure is explicitly included in the model some aspects of feature orientation cannot be reflected properly. In particular, we show that merging, overriding or extending bodies of methods yields problems. To overcome this deficiency, we relax the axioms and introduce the concept of an extended feature algebra. To clarify the idea and to underpin the relaxation we also extend the introduced model which can then handle code explicitly. Finally we discuss how additional operators can be introduced to capture even more properties of feature oriented programming formally. In particular, we present operators for merging, overriding and updating as they arise in code modification.

2. A STANDARD MODEL

Based on *feature structure trees* (FSTs), we give a first concrete model for feature algebra. The formal definition of feature algebras will be given in the next section. Fea-

ture structure trees capture the essential hierarchical module structure of a given program system (e.g. [3]). An example is given in Figure 1, where a simple Java class Base is described. For the present extended abstract we restrict ourselves to Java; examples of other feature oriented programming languages can easily described in a similar way.

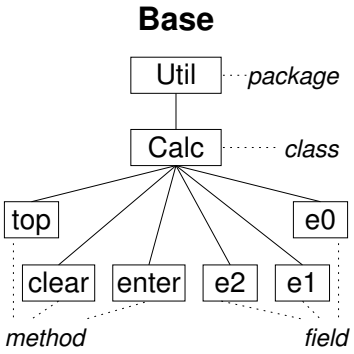


Figure 1: A simple JAVA-class as FST ([6, 3])

It is well known that certain labelled forests can be encoded using strings of node labels (e.g., [5]). We use forests rather than single trees in our description, since, in general, we deal with several classes.

Let Σ be an alphabet of node labels and, as usual, Σ^+ the set of all nonempty finite strings over Σ. Every such word can be thought of as the sequence of node labels along the unique path from a root in the forest to a particular node. In the sequel we will just write "path" instead of the lengthy "string of labels along a path". Note that this approach does not allow different roots with identical labels and no identical labels on the immediate descendants of a node. However, this is not a restriction.

A first model now represents a forest by all possible paths from roots to nodes. Since every prefix of the path leading to a node x corresponds to a path from the respective root to an ancestor of x, with a path also all its non-empty prefixes are paths in the forest. Therefore the set of all possible paths is prefix-closed. Note, however, that a *set* of paths forgets about the relative order of child nodes of a node, i.e., this model is suitable only for unordered trees.

EXAMPLE 2.1. *The FST of Figure 1 is encoded as the following prefix-closed set:*

$$\begin{aligned} Base =_{def} \{\ & Util,\ Util :: Calc,\ Util :: Calc :: top,\\ & Util :: Calc :: clear,\ Util :: Calc :: enter,\\ & Util :: Calc :: e0,\ Util :: Calc :: e1,\\ & Util :: Calc :: e2\ \}\ , \end{aligned}$$

where :: *is used to separate the elements of* Σ. *Of course all occurring names must be elements of the underlying alphabet, i.e.,* $Util, Calc, top, clear, enter, e0, e1, e2 \in \Sigma$. □

Conversely, one can (uniquely up to the order of branches) reconstruct a forest from the prefix-closed set of its paths.

We define $P\Sigma$ as the set of all prefix-closed subsets of Σ^+. Note that $P\Sigma$ is closed under set union. Based on this, feature tree superimposition can simply be defined as set union. Hence the order of combination does not matter and therefore addition is commutative and idempotent.

It is easy to show that $(P\Sigma, \cup, \emptyset)$ forms a monoid, i.e., $\cup$ is associative with $\emptyset$ as its neutral element, and, because of commutativity and idempotence of its addition operator $\cup$, also satisfies the axiom of distant idempotence, namely $A \cup B \cup A = B \cup A$ for $A, B \in P\Sigma$.

In addition to feature superimposition, feature algebra also comprises modifications which in the concrete model are tree rewriting functions.

It is easy to see that such functions can be used to model many different aspects of feature oriented programming and development. With respect to FSTs a modification might be the action of adding a new child node (adding a method to a class), of deleting a node (removing a method) or of renaming a node (renaming a class). As we will see, altering the contents of a leaf node (overriding and extending a method body) may lead to a problem.

A concrete tool for performing the operations of a feature algebra is *FeatureHouse* [1, 2]. It allows composing features written in various languages such as Java, C#, C, Haskell, and JavaCC. With the help of this tool we will show in Section 5 that the implementation and the axioms of feature algebra given below do not coincide when modifications are allowed to touch the code level. As long as the code is ignored, i.e., only the method interfaces or the like are considered to be modifiable things works fine.

This observation implies that, to achieve full congruence, either the theory has to be adapted or the implementation of *FeatureHouse* has to be changed. In Section 6 we introduce an extension of feature algebra that is designed to cover also features at code level.

3. FEATURE ALGEBRA

We now abstract from the concrete model of FSTs and introduce the structure of feature algebra. It was first presented by Apel, Lengauer, Möller and Kästner in [3]. There a number of axioms is selected that have to be satisfied by languages suitable for feature oriented software development. For the present paper we compact them and come up with the following definition. To focus on the main aspects we omit a discussion of the variants and alternatives described in the same paper.

A feature algebra comprises a set I of *introductions* that abstractly represent feature trees and a set M of *modifications* that allow changing the introductions. The central operations are the summation $+$ that abstractly models feature tree superimposition, the operator $\cdot$ that allows application of a modification to an introduction and the modification composition operator $\circ$.

Formally, a *feature algebra* is a tuple $(M, I, +, \circ, \cdot, 0, 1)$ such that

- $(I, +, 0)$ is a monoid satisfying the additional axiom of distant idempotence, i.e., $i + j + i = j + i$.
- $(M, \circ, 1)$ is a groupoid operating via $\cdot$ on I, i.e., $\circ$ is a binary inner operation on M and 1 is an element of M such that furthermore
  - $\cdot$ is an external binary operation from $M \times I$ to I
  - $(m \circ n) \cdot i = m \cdot (n \cdot i)$
  - $1 \cdot i = i$
- 0 is a right-annihilator for $\cdot$, i.e., $m \cdot 0 = 0$

- $\cdot$ distributes over $+$, i.e., $m \cdot (i + j) = (m \cdot i) + (m \cdot j)$

for all $m, n \in M$ and all $i, j \in I$.

On the introductions of a feature algebra, the *natural preorder* or *subsumption preorder* is defined by $i \leq j \Leftrightarrow_{def} i + j = j$; it is closely related to the subtyping relation $<:$ in the DEEP calculus of [15]. The *introduction equivalence* by $i \sim j \Leftrightarrow_{def} i \leq j \wedge j \leq i$. Finally, we define the *application equivalence* $\approx$ of two modifications m, n by $m \approx n \Leftrightarrow_{def} \forall\, i : m \cdot i = n \cdot i$. This is clearly an equivalence relation.

The model introduced in the previous section forms a feature algebra if a suitable set of tree rewriting functions is chosen as the set of modifications. The set has to be chosen carefully, since otherwise the functions might, e.g., violate the uniqueness conditions imposed on forests. The axiom $(m \circ n) \cdot i = m \cdot (n \cdot i)$ is satisfied by the usual definition of function composition: applying a composed function is equivalent to applying the single functions in sequence. Then the operator $\cdot$ coincides with function application and $\circ$ with function composition. Because of commutativity and idempotence of $\cup$ (which instantiates $+$ in that model), the natural preorder there actually is an order and coincides with the subset relation $\subseteq$.

An advantage of this particular abstract algebraic definition is that it contains only first-order equational axioms, i.e., it is predestined for automatic theorem proving. Since we have encoded feature algebra in Waldmeister [9][1], we skip the proofs. They can be found at a website [14].

LEMMA 3.1. *Assume i, j to be introduction sums and assume m, n, o to be modifications of a feature algebra.*

1. $0 \leq i$ *and* $i \leq 0 \Rightarrow i = 0$.
2. $+$ *is idempotent; i.e.,* $i + i = i$.
3. $\leq$ *is a preorder, i.e.,* $i \leq i$ *and* $i \leq j \wedge j \leq k \Rightarrow i \leq k$.
4. $i \leq i + j$ *and* $j \leq i + j$.
5. $i \leq k \wedge j \leq k \Rightarrow i + j \leq k$.
6. $+$ *is quasi-commutative w.r.t.* $\sim$, *i.e.,* $i + j \sim j + i$.
7. $(m \circ (n \circ o)) \cdot i = ((m \circ n) \circ o) \cdot i$.
8. $(m \circ 1) \cdot i = (1 \circ m) \cdot i = m \cdot i$.

Meanings and relevance of Parts (1)–(3) are straightforward. Part (4) says that addition determines an upper bound with respect to the natural preorder. Part (5) shows that the sum is even a least upper bound. Parts (7) and (8) show that, up to application equivalence, $\circ$ is associative and 1 is its neutral element, i.e., $(m \circ (n \circ o)) \approx ((m \circ n) \circ o)$ and $(m \circ 1) \approx (1 \circ m) \approx m$.

4. ANOTHER STANDARD EXAMPLE

Since in certain applications the relative order of the immediate successor nodes in a tree matters, we now present a second model that reflects forests of ordered labelled trees. It uses the fact that all paths in a tree can be recovered from the maximal ones that lead from roots to leaves by forming their prefix closure. It should be noted here that the maximal paths can be viewed as atomic introductions in the sense of [3]. This could have been exploited already in the previous model, but would have led to a much more complicated definition of tree superimposition. While an unordered forest can be represented as the finite *set* of its maximal paths, for an ordered one we use finite *lists* of such paths. To make the representation unique, we have to restrict ourselves to lists that are prefix-free, i.e., lists l that with a path p do not contain a proper or improper prefix of p elsewhere in l. In particular, such lists are repetition-free. Like the previous model, this does not admit different roots with identical labels and no identical labels on immediate descendants of a node.

EXAMPLE 4.1. *The FST of Figure 1 is encoded as the following prefix-free list:*

$$\begin{aligned} Base =_{def} [\ & Util :: Calc :: top,\ Util :: Calc :: clear,\\ & Util :: Calc :: enter,\ Util :: Calc :: e2,\\ & Util :: Calc :: e1,\ Util :: Calc :: e0\ ]\ . \end{aligned}$$

□

Superimposition $+$ is now defined inductively over the length of the first list:

- The empty list does not effect another list of paths:

$$[\,] + [q_1, \ldots, q_n] =_{def} [q_1, \ldots, q_n]$$

- A singleton list $[p]$ is added to an existing list by replacing existing prefixes of it:

$$[p] + [q_1, \ldots, q_n] =_{def} \begin{cases} [q_1, \ldots, q_n] & \text{if } p \text{ is a prefix of some } q_i \\ [q_1, \cdots q_{i-1}, p, q_{i+1}, \ldots, q_n] & \text{if } q_i \text{ is a prefix of } p \\ [p, q_1, \ldots, q_n] & \text{otherwise} \end{cases}$$

- For longer lists we set

$$\begin{aligned} &[p_1, \ldots, p_m, p_{m+1}] + [q_1, \ldots, q_n] =_{def}\\ &\quad [p_1, \ldots, p_m] + ([p_{m+1}] + [q_1, \ldots, q_n]) \end{aligned}$$

We define $L\Sigma$ as the set of all prefix-free lists of elements of Σ^+. It is easy to show that $(L\Sigma, +, [\,])$ forms a (non-commutative) monoid that additionally satisfies the axiom of distant idempotence; its natural preorder reflects list inclusion and the associated equivalence relation is permutation equivalence, i.e., equality up to a permutation of the list elements. Also in this model, modifications are just rewriting functions with the same operations as before.

In both algebras $P\Sigma$ and $L\Sigma$ distant idempotence models the fact that duplicating a feature has no effect. Hence idempotence seems of central interest in feature algebra. However, in the next section we will show that the axiom of distant idempotence yields problems in a model that considers more details.

5. THE LOST IDEMPOTENCE

As mentioned in the previous sections, distant idempotence (and hence idempotence), i.e., the fact that duplicating a feature has no effect, was of central interest in feature

[1] In contrast to [4], we use Waldmeister instead of Prover9 since it can handle multiple sorts. For feature algebra we use the two sorts M and I.

algebra. In [4], it is stated that languages and tools for feature combination usually have the idempotence property.

This works fine as long as a feature only contains the name and not its implementation. At the code level this property does not hold any longer. We illustrate this behaviour by a Java program.

EXAMPLE 5.1. *Consider a Java method* `foo` *given by*

```
void foo(int a) {
  a++;
  original(a);
}
```

When used in a feature superimposition, it updates a previous definition of `foo`*; the pseudo-statement* `original(a)` *inserts the original body. We assume further that* `foo` *is a method of the class* `Bar`. □

To integrate code into an FST, each terminal node has to be extended. According to this, we have also to extend the prefix-closed elements of the set $P\Sigma$. This is done as follows: Each letter (element of Σ) at the end of a maximal path is extended with code. This extension preserves that prefixes of paths are legal paths again. The following example should clarify the main idea; an abstract and more precise definition will be given below.

EXAMPLE 5.2. *With this explanation, the code of the previous example can be written as*

Bar::foo

```
void foo(int a) {
  a++;
  original(a);
}
```

To shorten the notation we write $\texttt{Bar} :: \texttt{foo}[A]$ *where* A *is an abbreviation for the complete code contained in the box.* □

As mentioned before, feature algebra was introduced as a formal treatment of FOSD and is intended to model *FeatureHouse* at an abstract level. If, however, two occurrences of the same feature appear in one program the code parts have to be merged and hence the order of combination does matter, since code parts of the are overwritten and/or updated. We skip the details how *FeatureHouse* merges code and applies overriding. Instead we illustrate the situation by an example.

EXAMPLE 5.3. *Using FeatureHouse leads to the following result:*

$$\texttt{Bar} :: \texttt{foo}[A] \oplus \texttt{Bar} :: \texttt{foo}[A]$$

Bar::foo

```
void foo(int a) {
  a++;
  a++;
  original(a);
}
```

$$= \quad \neq \quad \texttt{Bar} :: \texttt{foo}[A]$$

where $\oplus$ *is the feature combination of FeatureHouse. In particular,* $\oplus$ *is not idempotent and therefore the axiom of distant idempotence does not hold.* □

This short example and this short application of *FeatureHouse* show that idempotence is not satisfied in general in the setting of FOSD. Moreover, either feature algebra is not the formal model for *FeatureHouse* or *FeatureHouse* does not follow the theoretical foundations introduced by the algebraic structure.

This section provided only a brief description and focussed on some parts of *FeatureHouse* and feature algebra which lead to discrepancies. It was not the intention to explain every fact of *FeatureHouse* and feature algebra in detail. The interested reader is referred to the references [1, 3].

6. EXTENDED FEATURE ALGEBRA

We have shown that the axioms of distant idempotence and hence also standard idempotence do not hold when arguing at code level. The remainder of the paper presents some ideas how to solve the described problems.

To model code-level behaviour at an abstract level we extend feature algebra by a third type C of *code fragments*.

We define the structure of an *extended feature algebra* as a tuple $(M, I, C, +, \circ, \cdot, |, 0, 1)$ with the following properties for all $m, n \in M$, $i, j \in I$ and $a, b, c \in C$:

- We consider pairs (i, c) where i is an introduction corresponding to a maximal path in the forest under consideration and c is the code fragment contained in the leaf at the tip of that path. We denote (i, c) by $i[c]$.[2] The set of all these pairs is denoted by $I[C]$.
- $(C, |)$ is a semigroup in which $|$ is an update or override operation (see below),
- $(I[C], +, 0)$ is a monoid satisfying $i[a] + j[c] + i[b] = j[c] + i[a|b]$,
- $(M, \circ, 1)$ is a groupoid operating via $\cdot$ on $I[C]$,
- 0 is a right-annihilator for $\cdot$ and
- $\cdot$ distributes over $+$.

The original definition of a feature algebra can be retrieved by choosing C as a set containing only one single element (the empty code fragment).

The operation $|$ can be seen as an update. In the previous section, $|$ merged code fragments. In the next section we will discuss this operation in our concrete models. Note that we have modified the axiom of distant idempotence: adding a feature a second time updates the earlier instance of that feature rather than just ignoring it.

Unfortunately, we cannot define a natural preorder on an extended feature algebra. This is due to the lack of idempotence. Hence the counterpart of Lemma 3.1 reduces to

LEMMA 6.1. *Assume* i, j *to be introductions,* m, n, o *to be modifications and assume* c *to be a code fragment of an extended feature algebra. Then*

1. $(m \circ (n \circ o)) \cdot i[c] = ((m \circ n) \circ o) \cdot i[c]$,
2. $(m \circ 1) \cdot i[c] = (1 \circ m) \cdot i[c] = m \cdot i[c]$.

[2] This fits well with the notation of the example of the previous section.

To overcome the deficiency of the missing preorder we can define two different relations:

$$i[a] \leq_r j[b] \quad =_{def} \quad \exists\, k[c] \in I[C] : i[a] + k[c] = j[b] \ ,$$
$$i[a] \leq_l j[b] \quad =_{def} \quad \exists\, k[c] \in I[C] : k[c] + i[a] = j[b] \ .$$

This implies immediately the following

LEMMA 6.2.

1. $\leq_l$, $\leq_r$ *are preorders*
2. $0 \leq_l i$, $0 \leq_r i$

Up to now we do not know which of the orders should be preferred. A further investigation of properties as well as the interaction of both preorders will be part of future research (cf. Section 8).

7. EXTENDING THE MODELS

In Section 5 we pointed out that *FeatureHouse* merges and updates code. Therefore a formal model should also reflect this behaviour. Unfortunately this does not hold for the models presented in Section 2, which led to our extension by code fragments.

In this section, we show how to define the update operator | in our concrete models.

In particular, we will identify the "common part" of two given implementations of the same feature oriented program. Based on the common part one can determine which part of a method body has to be overridden and which part has to be preserved. Of course these calculations highly depend on the respective language and have to follow exact rules. In Java, for example, *FeatureHouse* simply overrides declarations and functions as long as the keyword `original` does not occur in the code.[3] For a detailed description we refer again to [1].

To model such behaviour we define *abstract interfaces* for each Java method. Whereas a general Java element may contain arbitrary (legal) programming constructs, abstract interfaces may contain only the types of the corresponding Java parts and "forgets" the remaining bodies, initialisations etc. We illustrate this behaviour by an example.

EXAMPLE 7.1. *On the left hand side there is a simple Java method while its abstract interface appears on the right hand side.*

```
int min5(int a) {
  int b=5;
  if(a<b) return a;
  else return b;
}
```

```
int min5(int a) {
  int b;
}
```

The typing of the local variable `b` *appears, since its declaration may be overwritten during a feature combination.* □

A precise definition of the abstract interface will need to reflect also nested scopes etc. The use of abstract interfaces may yield invalid Java code (e.g., `return` statements are omitted). This does not matter, though, since it will only be employed to identify the "common part".

Let again C be the set of possible code fragments and $T \subseteq C$ the set of the corresponding abstract interfaces. The function that determines the abstract interface for a given Java code is denoted by $ai : C \to T$. Next we define two functions

$$\begin{aligned} \ddagger, - \quad &: \quad \mathcal{P}(C) \times \mathcal{P}(T) \to \mathcal{P}(C) \\ X \ddagger U \quad &= \quad \{x \in X \,|\, ai(x) \in U\} \\ X - U \quad &= \quad \{x \in X \,|\, ai(x) \notin U\} \ . \end{aligned}$$

The restriction operator $\ddagger$ determines for a set X of code fragments the ones whose corresponding abstract interfaces lie in the given subset $U \subseteq T$, while the operator $-$ selects its relative complement.

To define the update function | we need to lift the function ai to sets of code fragments by

$$ai(X) \ =_{def} \ \{ai(x) \,|\, x \in X\} \ .$$

Then

$$X|Y \ =_{def} \ (Y - ai(X)) \cup X \ .$$

This means that all "old" definitions of elements in Y that are redefined in X are discarded and replaced by the ones in X; moreover, all elements of X not mentioned in Y are added. It should be noted that ai and | are closely related to the interface operator $\uparrow$ and the asymmetric composition $\&*$ in the DEEP calculus of [15].

It turns out that this rather concrete definition can be lifted to the same level of abstraction as that of our feature algebra, which again opens the possibility for automated verification. The key is the observation that $ai(X)$ is the least set that leaves X unchanged under the selection operation $\ddagger$:

$$X = X \ddagger U \ \Leftrightarrow \ ai(X) \subseteq U$$

In fact, since $X \ddagger U \subseteq U$ holds anyway by definition, this can be relaxed to

$$X \subseteq X \ddagger U \ \Leftrightarrow \ ai(X) \subseteq U \ .$$

Mathematically, this is known as a *Galois connection* (e.g. [8, 11]). Also, ai behaves in many respects like the abstract codomain operator of [12]. The definition of | is similar to the ones based on relations or semirings with domain (e.g. [16, 13]). These correspondences allow us to re-use a large body of well-known theory — another advantage of an abstract algebraic view.

Let us detail this a bit more. We may abstract the set C of code fragments to a Boolean algebra L and the set T of abstract interfaces to a subalgebra N of L. Then the above functions can be characterised and generalised using the following axioms:

$$\begin{array}{l|l} (a+b) \ddagger p = a \ddagger p + b \ddagger p & (a+b) - p = (a-p) + (b-p) \\ a \ddagger 0 = 0 & a - 0 = a \\ a \ddagger (p+q) = a \ddagger p + a \ddagger q & a - (p+q) = (a-p) - q \\ & \qquad = (a-q) - p \\ 0 \ddagger p = 0 & 0 - p = 0 \end{array}$$

$$\begin{aligned} a \leq a \ddagger p \quad &\Leftrightarrow \quad \overline{a} \leq p \\ a|b \quad &= \quad (b - \overline{a}) + a \end{aligned}$$

where $a, b \in L$, $p, q \in N$ and $+$ denotes the supremum of L, $\leq$ its order, 0 the least element and $\overline{a}$ is the abstract counterpart of $ai(a)$.

Note that this section only gives the main ideas how to model the update operation in the abstract setting of an extended feature algebra. The work presented is part of ongoing work and will be investigated in much more detail (see the next section).

[3] There are some exceptions.

8. CONCLUSION AND OUTLOOK

The present paper is based on earlier work by Apel, Lengauer, Möller and Kästner [3]. They introduced a formal model to capture the commonalities of feature oriented software development such as introductions, refinements and quantification. We have defined a concrete model for feature algebra and have illustrated that the axioms of feature algebra are fine as long as one does not consider feature oriented programming at code level. Otherwise not all aspects of feature orientation can be modelled. To remedy this, we have introduced the structure of an extended feature algebra which generalises the original definition. To clarify the idea we have also extended the introduced models correspondingly. Finally we sketched how additional operators can be introduced to capture even more properties of feature oriented programming like updating or overriding.

This extended abstract is a further step towards an algebraic theory that covers all aspects of FOSD. Of course, all introduced operators like update need further investigation; in particular Section 7 reports about ongoing work. On the one hand more properties need to be derived; on the other hand it has to be checked whether the extension adequately covers the essential properties of FOSD, in particular, the merging of code fragments. If this turns out not to be the case, the extended feature algebra will need further modification.

Acknowledgement.
We are grateful to Han-Hing Dang, Roland Glück and the anonymous referees for fruitful comments.

9. REFERENCES

[1] S. Apel, C. Kästner, and C. Lengauer. FeatureHouse: Language-independent, automated software composition. In *31th International Conerence on Software Engineering(ICSE)*, pages 221–231. IEEE Press, 2009.

[2] S. Apel and C. Lengauer. Superimposition: A language-independent approach to software composition. In C. Pautasso and É. Tanter, editors, *Software Composition*, volume 4954 of *Lecture Notes in Computer Science*, pages 20–35. Springer, 2008.

[3] S. Apel, C. Lengauer, B. Möller, and C. Kästner. An algebra for features and feature composition. In *AMAST 2008: Proceedings of the 12th international conference on Algebraic Methodology and Software Technology*, volume 5140 of *Lecture Notes in Computer Science*, pages 36–50. Springer, 2008.

[4] S. Apel, C. Lengauer, B. Möller, and C. Kästner. An algebraic foundation for automatic feature-based program synthesis and architectural metaprogramming. *Science of Computer Programming*, 2009. (to appear).

[5] F. Baader and T. Nipkow. *Term Rewriting and All That*. Cambridge University Press, 1999.

[6] D. Batory. From implementation to theory in product synthesis. *ACM SIGPLAN Notices*, 42(1):135–136, 2007.

[7] D. Batory and S. O'Malley. The design and implementation of hierarchical software systems with reusable components. *ACM Transactions Software Engineering and Methodology*, 1(4):355–398, 1992.

[8] G. Birkhoff. *Lattice Theory*. American Mathematical Society, 3rd edition, 1967.

[9] A. Buch, T. Hillenbrand, and R. Fettig. Waldmeister: High Performance Equational Theorem Proving. In J. Calmet and C. Limongelli, editors, *Proceedings of the International Symposium on Design and Implementation of Symbolic Computation Systems*, number 1128 in Lecture Notes in Computer Science, pages 63–64. Springer, 1996.

[10] K. Czarnecki and U. Eisenecker. *Generative Programming: Methods, Tools, and Applications*. 2000.

[11] B. A. Davey and H. A. Priestley. *Introduction to lattices and order,*. Cambridge University Press, 2nd edition, 2002.

[12] J. Desharnais, B. Möller, and G. Struth. Kleene algebra with domain. *ACM Transactions on Computational Logic*, 7(4):798–833, 2006.

[13] T. Ehm. Pointer Kleene algebra. In R. Berghammer, B. Möller, and G. Struth, editors, *RelMiCS*, volume 3051 of *Lecture Notes in Computer Science*, pages 99–111. Springer, 2004.

[14] P. Höfner. Database for automated proofs of Kleene algebra. `http://www.dcs.shef.ac.uk/~georg/ka` (accessed September 25, 2009).

[15] D. Hutchins. Eliminating distinctions of class: Using prototypes to model virtual classes. In P. L. Tarr and W. R. Cook, editors, *Proceedings of the 21th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2006*, pages 1–20. ACM Press, 2006.

[16] B. Möller. Towards pointer algebra. *Science of Computer Programming*, 21(1):57–90, 1993.

Dead or Alive: Finding Zombie Features in the Linux Kernel*

Reinhard Tartler, Julio Sincero, Wolfgang Schröder-Preikschat, Daniel Lohmann
{tartler, sincero, wosch, lohmann}@cs.fau.de

Friedrich-Alexander-University Erlangen-Nuremberg

ABSTRACT

Variability management in operating systems is an error-prone and tedious task. This is especially true for the Linux operating system, which provides a specialized tool called *Kconfig* for users to customize kernels from an impressive amount of selectable features. However, the lack of a dedicated tool for kernel developers leads to inconsistencies between the implementation and the variant model described by *Kconfig*. This results in real bugs like features that cannot be either enabled or disabled at all; the so called *zombie* features.

For both in the implementation and the variant model, these inconsistencies can be categorized in *referential* and *semantic* problems. We therefore propose a tool approach to check the variability described by conditional compilation in the implementation with the variant model for both kinds of consistency. Our analysis of the variation points show that our approach is feasible for the amount of variability found in the Linux kernel.

Categories and Subject Descriptors

D.4 [**Operating Systems**]: Organization and Design; D.2.16 [**Software Engineering**]: Configuration Management; D.3.4 [**Programming Languages**]: Preprocessors

General Terms

Design, Language, Tool Support

Keywords

Software Product Lines, Features, Preprocessor, Linux

1. INTRODUCTION

*This work was partly supported by the German Research Council (DFG) under grants no. SCHR 603/7-1 and SCHR 603/4

FOSD'09, October 6, 2009, Denver, Colorado, USA.

Operating systems provide no business value of their own. Their sole purpose is to ease the development and execution of applications on some hardware platform, that is, to serve application developers and application users with a virtual machine layer that provides the "right" set of features for *their* particular problem domain. Of course, this pays off only if the operating-system itself is *reusable* for many different applications and hardware platforms. Historically, this led to the idea of configurable *system families*, in which each family member subsumes a particular set of features. In fact, much of the work of the software-engineering pioneer's from the 70s was motivated by practical problems that stem from the feature variability of configurable operating-system families [5, 15, 6].

Today, the number of configurable features offered by many operating systems is still an order of magnitude higher than the variability we typically find in software product lines and software families from other domains: The feature model of the (very small) PURE embedded operating system, for instance, already offers more than 250 configurable features [2]; eCos [13], which targets the same domain, provides more than 750 features. However, the Linux kernel, which is the subject of this paper, can be configured by even more than 8000 (!) configuration options [17].

It should be clear that both, kernel hackers and end users, have to be supported by extra means and tools for *variability management* to handle this impressive amount of features. With the graphical configuration editors build around the *Kconfig* tool, end-user support is already very reasonable. For kernel hackers, however, variability management is an error-prone and tedious task. The result are bugs and *zombie*-features that are still presented in *Kconfig*, but not in the code or vice versa.

1.1 Variability Management in Linux

The variability management techniques employed in the Linux kernel can be divided into three levels:

Model Level. The *Kconfig* tool set was especially written to support the modeling of features and interdependencies of the Linux kernel. It provides a language to describe a variant model consisting of features (referred to as *config options*) together with their constraints and dependencies. Modularization of the variant model is supported by an inclusion mechanism. In Linux kernel version `2.6.30`, a total of 534 `Kconfig` files are employed, consisting of 88,112 lines of code that describe 8063 features and their dependencies. In many respects, the resulting variant model can be compared

to feature models known by the software product-line community [18].
The user configures a Linux kernel by selecting features from this model. During the selection process, the *Kconfig* configuration utility implicitly enforces all dependencies and constraints, so that the outcome is always the description of a valid variant. Technically, this description is given as a C-style header file that defines a `CONFIG_xxx` preprocessor macro for every selected feature.

Generation Level. Coarse-grained variability is implemented on the generation level. The compilation process in Linux is controlled by a set of custom scripts called *Kbuild* that interpret a subset of the `CONFIG_xxx` flags and drive the compilation process by selecting which compilation units should be compiled into the kernel, compiled as a loadable module, or not compiled at all.

Source Code Level. Fine-grained variability is implemented by conditional compilation using the C preprocessor. The source code is annotated with preprocessor directives (like `#ifdef CONFIG_xxx` or `#if(CONFIG_xxx ...)`), which are evaluated in the compilation process. This is the major variability mechanism used in the Linux kernel.

Whereas in other domains the set of features is usually the outcome of a top-down domain analysis (*requirement-motivated features*), the majority of features we find in configurable operating systems are usually the result of a bottom-up design and implementation process, beginning on the level of hardware-abstractions up to the kernel APIs (*implementation-motivated features*). This is particularly true in Linux, which is a very source-code–centric project. New or improved features are implemented first; later on they are assigned a `CONFIG_xxx` symbol which is, together with their dependencies, integrated into or updated in the *Kconfig* model. This, however, is a manual and tedious task that nevertheless requires the skills of an experienced kernel hacker.

1.2 Problem Statement

It is not difficult to imagine that this process leads to inconsistencies between the *Kconfig* representation of features and their dependencies as seen by the user who configures a Linux kernel, and the implementation.

It is clear that the *Kconfig* tool cannot solve variability management problems satisfactorily for kernel developers, as it has no access to the *use* of its *config options* in the code base. Likewise, the C Preprocessor is not able to detect inconsistencies of *config options* during parsing because it has no capabilities for interpreting the *Kconfig* model.

Undoubtedly, the variability described in the *Kconfig* files and in the source code are conceptually interconnected and have to be kept *consistent*. However, there is a gap between the various tools that are involved during the configuration and compilation phase of the Linux kernel. This gap has to be closed in order to detect existing bugs and avoid new ones when new features are added or existing features are refactored. This is especially important because changes (new additions or refactoring) at both sides (*Kconfig* model or code base) may potentially break consistency.

1.3 Our Contribution

In this paper we introduce a set of *conditions* that have to be asserted between code base and the *Kconfig* model in order to preserve consistency. We also confirm that this is a real problem by analyzing the current code base of the Linux kernel and show real bugs. We also expound that such bugs are introduced due to inadequate tool support. Moreover, we sketch the requirements of a tool to detect such inconsistencies, and, therefore, is able to close the gap between the variability at *model level* and *source code level*.

2. PROBLEM ANALYSIS

While analyzing the Linux family model we discovered that the implementation in form of C source code and the family model described in *Kconfig* show obvious inconsistencies. In summary, we identify two dimensions of consistency.

Referential consistency is categorized by:

1. every reference to a configuration item in the source code is referenced in the *Kconfig* variant model
2. every *Kconfig* option from the variant model is referenced from the implementation in the source code

Furthermore, *semantic consistency* can be accounted by:

3. every single configuration item in the source code is selectable in the *Kconfig* variant model
4. the *Kconfig* variant model covers all `#if` branches for all conditional blocks that use more than one configuration item

Fragments in the source code or in *Kconfig* that violate one or more of these consistency conditions result in features that can never be enabled or disabled. We therefore call features, which are implemented by fragments that are either *always* dead or alive *zombie features*.

In this paper, we focus on variation points that are implemented by *configuration-controlled* conditional compilation. Configuration controlled means that only conditional blocks that are affected by the configuration selection are considered. In Linux that means preprocessor macros beginning with the string `CONFIG_`. In theory, these macros should only be set in generated, configuration-derived files. In practice, this rule is not strictly enforced, so that estimations need to be made carefully.

As of version 2.6.23, the Linux source tree contains a script `scripts/checkkconfigsymbols.sh` that is intended to test the source code against the *Kconfig* model with respect to *referential integrity* regarding `CONFIG_xxx` symbols. However, this script is obviously not employed by the Linux maintainers, as a simple test run reports over **760** unresolvable references in kernel version 2.6.30. We modified the script to avoid false positives by only considering unresolvable references that appear in preprocessor directives (i.e., only in lines that start with `#if` or `#elif`). Even though this is most probably too strict and we thereby miss some legitimate problems, the number of unresolvable references only went down to **360**. Further inspection of the Linux code base shows that **206** macros, which start with the `CONFIG_` prefix, are being defined or undefined with the preprocessor. From these macros, only **39** are defined within *Kconfig*, this means that we estimate at least **321** real issues that need to be investigated!

This number is still calculated conservatively. A first analysis of the findings reveal several obvious bugs, such as the misspelling of `CONFIG_CPUMASKS_OFFSTACK` in the file `include/linux/irq.h` (most probably `CONFIG_CPUMASK_OFFSTACK` was meant), or the item `CONFIG_CPU_HOTPLUG` in the file `kernel/smp.c`, which should probably read `CONFIG_HOTPLUG_CPU`.[1]

Even though these number of inconsistencies already sounds pretty alarmingly, they most probably cover only the tip of the iceberg. Besides additional inconsistencies with respect to *referential integrity* that arise from *Kconfig* items that are not present in the source code, we can also expect inconsistencies between the encoding of feature dependencies and feature interactions in *Kconfig* and in the source code.

3. SOLUTION OUTLINE

It is obvious that the simple approach taken by the `scripts/checkkconfigsymbols.sh` does only cover checking for *referential consistency* from the implementation to the variant model. However, we require detecting violations of all four conditions (i.e. both *referential* and *semantic consistency*), with satisfying accuracy.

Conditional compilation is implemented by preprocessor directives that order conditional blocks in a defined order, and is straightforward to analyze. However quantifying *configuration-controlled* variability is more challenging. Consider the following source code excerpt taken from the file `include/linux/init.h`:

```
#ifndef _LINUX_INIT_H
#define _LINUX_INIT_H
[...]
#if defined(MODULE) || defined(CONFIG_HOTPLUG)
#define __devexit_p(x) x
#else
#define __devexit_p(x) NULL
#endif

[...]
#endif /* _LINUX_INIT_H */
```

This source code snippet shows a typical C header. The very first preprocessor statement in this file is a technique known as "`#include`-guard", so that multiple inclusions of this file do not cause the actual contents to be evaluated multiple times by the compiler. In any case, these kinds of blocks are clearly not *configuration controlled* and therefore, must not be counted. Next, this header defines a qualifier `__devexit_p(x)` whose exact definition depends on both a configuration dependent variable `CONFIG_HOTPLUG` as well as another macro named `MODULE`. This macro can be totally unrelated to the *Kconfig selection*. Therefore, our tool does consider this conditional block as such, but for analyzing the variability of the compilation unit, only the macro `CONFIG_HOTPLUG` is considered.

However, not all conditional blocks contribute to *configuration-controlled* variability. Consider the following code snipped taken from the file `kernel/printk.c`:

```
static int __init console_setup(char *str)
{
[...]
#ifdef __sparc__
        if (!strcmp(str, "ttya"))
                strcpy(buf, "ttyS0");
        if (!strcmp(str, "ttyb"))
                strcpy(buf, "ttyS1");
#endif
[...]
}
```

The purpose of this conditional block is portability of the file to the Sparc architecture. While this is an important concern, it is not handled by the means of *Kconfig* and therefore cannot be controlled by the user.

The preprocessor is used in this source file to include the previously shown header textually before passing the composed text to the compiler. In order to calculate the variability of the expanded compilation unit, the `#include` directive needs to be expanded. This allows us to consider both cases: The variability of the source file – the developer's view – and the variability of the compilation unit – the view of the compiler.

Interestingly, in Linux the configuration selection is not referenced explicitly in any source file. Instead, the configuration is implicitly present with a *forced-*`#include` technique as implemented by the `-include` compiler command-line option of GCC. This technique is used in all compilation units used during the compilation phase of the Linux kernel. Tools for evaluating the Linux source code therefore have to adopt this technique as well.

In this example we identified two `#ifdef`-blocks that are configuration dependent. The `#ifdef`-statement in the main source file does reference an identifier with the substring `CONFIG_`. However, it does not follow the convention that configuration items in Linux must **begin** with that prefix. Moreover the `#if`-statement in the header also contains an `#else`-block, which we count as an extra block.

A tool that reliably detects these inconsistencies requires a *global view* on all variation points in both the source code (the declarations of *conditional blocks*) and the *Kconfig* family model. As a first step, our framework therefore builds an *Implementation Variability Database* by scanning the source code. After scanning the complete kernel tree, the database contains all configuration-dependent conditional-blocks from the *implementation* of Linux. This is depicted in the lower part of Figure 1.

This database is essentially a fact database. In order to query such a database efficiently, first-order logic provides an appropriate language to create queries that allow drawing further conclusions. We therefore use the `crocopat` tool for relational programming [3] for this task. `crocopat` uses the *rigi standard format* from the rigi [20] reverse-engineering suite as input format. With `crocopat`, we can trivially do prolog-like queries on the *Implementation Variability Database* of any sort.

From the *Kconfig* files we extract the dependencies and constraints into a second database, the *Family Model Variability Database*. For this we need to parse the existing *Kconfig* files in Linux and analyze the dependencies and constraints. In order to be able to do queries and selections, we use the *rigi standard format* again so that `crocopat` can be reused. This is sketched in the upper part of figure 1.

[1]We have reported these inconsistencies as potential bugs to the Linux community and are awaiting a confirmation.

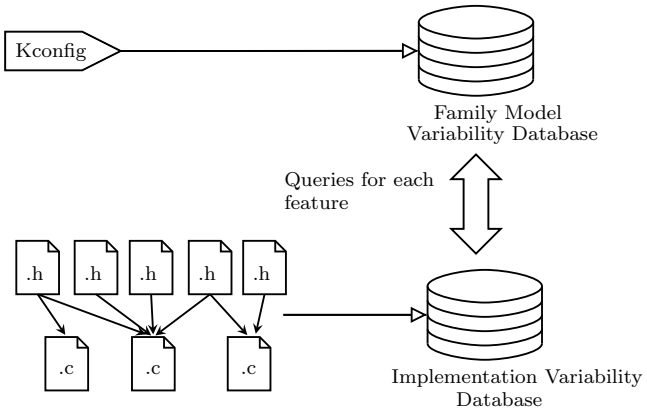


Figure 1: Our tool approach

In order to build the *Implementation Variability Database* we will calculate the *variability* of individual expanded compilation units. In this context *variability* means all possible different token streams (the input for syntactic parsing) that can be generated by the C Preprocessor (`CPP`) when preprocessing a compilation unit. When the expression of a `CPP` directive evaluates to true, the corresponding code will be read and inserted into the resulting token stream. If the expression evaluates to false, this code block is skipped and not included in the token stream. It means that in theory, a file with n different conditional directives (`#if`, `#elif`, etc.) might consist of 2^n different combinations.

Fortunately, many of these configurations cannot be composed in practice. For example, the conditional blocks defined by an `#if` directive and its corresponding `#else` will never be enabled at the same time, because of the exclusive semantic of `#if-#else` blocks. In order to calculate the exact variability of each compilation unit, our tool generates a formula like $f\colon (x_1, \ldots, x_n) \to \{0, 1\}$ where $x_1, x_2, \ldots, x_n$ are the *Kconfig* symbols used in `CPP` statements. This formula is basically the conjunction of the condition of each conditional block. These conditions are in fact the conjunction of the expression where the *Kconfig* symbols are used (e.g. `CONFIG_SMT && CONFIG_X86`) with the structural constraints like nested blocks (implications of the form ($child \Rightarrow parent$)). Another type of constraint is imposed by the semantics of `#if-#elif-#else` block groups, where for each block $(B_1, B_2, \ldots, B_n)$ of such group a dependency of the form $B_1 \Rightarrow \neg(B_2 \vee B_3 \vee \ldots \vee B_n)$ is required.

After constructing such a boolean function, it can be translated into a *binary decision diagram* (BDD) so that further analysis, like the calculation of the truth table, can be performed very efficiently. The lines in the truth table of such a boolean function that evaluate to true provide the set of valid configurations of a compilation unit. We can then use this set of valid configurations and crosscheck with the *Kconfig* dependencies in order to discover combinations of features that are allowed in the code base but not in the variant model, the so called, *zombie* features.

Having both databases available will allow us to check *referential integrity* of configuration items. In order to check *semantic integrity*, we transform the feature dependencies of the *Family Model Variability Databases* into BDDs that can be queried for satisfiability very efficiently. This way, checking satisfiability for each single configuration option is just a query against the BDD.

Checking semantic integrity with `#ifdef`-blocks that depend on more than one *configuration option* is more challenging. In a first step, we use the *Implementation Variability Database* to identify all conditional blocks with more than one configuration item and obtain the exact expression used in the conditional block. The challenge here is that conditional blocks may be influenced by configuration items both *explicitly* and *implicitly*. With *explicit influence* we mean that the *Kconfig symbol* appears literally in the expression of the preprocessor directive. *Implicit influence* happens either by nested `#ifdef` directives or when the `#define` preprocessor directive is used in a conditional block to define another configuration item. The framework must consider all configuration items (both implicit and explicit) for each conditional block and calculate on this basis the conditional-compilation *path-coverage*. Blocks that cannot be reached in any configuration are then in violation with the *semantic consistency* condition.

4. DISCUSSION

In the previous section, we have outlined our proposed tool that will be part of a greater framework to assist managing and verifying the variability in the Linux kernel. We hope that our tool will be adopted by the various Linux communities involved with variability management and issues that arise from it.

Especially targeted for driver developers, our framework would be able to show what configuration derived variability is actually used by a device driver. This would highlight the variability points introduced for example by driver developers, but also indicate interaction with other features that might have not been considered (yet) during the implementation of the driver. Depending on these additional variation points, this can indicate that additional test cases need to be considered.

Similarly, subsystem maintainers could use our tool during reviews and integrations. While inspecting the Linux Guidelines for patch submission and review[2] it turns out that **9** out of **24** points deal with *Kconfig* related issues. These issues are very hard to test and review; our framework can assist here with visualizing and verifying the additional variation points.

While we are convinced that our framework will be useful for kernel maintainers, we need to consider if our approach scales with the amount of variability in Linux. With Linux, we are facing a variant model of about **8000** features. It is well known that the size of BDDs is very sensitive to the number and order of its variables, which may lead to insufficient memory problems. However, our preliminary results clearly show that the variability is not uniformly distributed across the Linux source code, but *variability hot spots* can be identified easily. Therefore, we will work on a per compilation-unit basis in order to keep the BDDs reasonably sized.

A first analysis with a self written tool based on sparse [10], a framework for static analysis written for the Linux kernel, shows that less than **10%** of all files of the Linux kernel use more then **2** different *Kconfig* symbols. When considering

[2]as found in the file `Documentation/SubmittingPatches`

expanded compilation units, we see that more than **85%** of all compilation units have at most **350** different symbols in them. It is clear that we must also consider *Kconfig symbols* that are not only explicitly named in a compilation unit but come into effect indirectly. This can happen for example when a compilation unit *overrides* a configuration item with the `#define` preprocessor statement. Moreover, we need to compute a *global variable order* so that partial computation results can be reused. According to Mendonca et. al. [14], the largest feature models that can be handled today have about 2000 features and 400 extra constraints. Still, we expect that most compilation units in Linux will not exceed these limits, if any.

Is this a language problem?

Could the problem have been avoided in the first place? In many cases the usage of the C-Preprocessor is held responsible for maintenance problems in large software projects [19]. Would it be feasible to avoid the preprocessor in Linux? For Linux, the answer is *no*. Operating systems need modularization that is finer grained than provided by the plain C language. This fact was already known during the design and implementation of the C programming language [9], so the C Preprocessor became part of every implementation of C. With the conditional compilation feature, fragments of a program can be modularized at a *sub-statement* level. However, we identify two main issues with the approach taken by the preprocessor: a) conditional blocks cannot take any context into account, and b) the blocks are declared anonymously and cannot be referenced from anywhere.

The first point is necessary for any form of generic implementation. While conditional blocks cannot handle this directly, the common workaround for this limitation is to declare a preprocessor macro that takes parameters and declare multiple, alternate implementations of the macro. Later, in the C implementation, the macro is called like a function – however parameters are bound *by name* to the macro code. This workaround has limitations: First, the macro needs to be declared in a single line, although line continuations with the backslash (\) character are allowed. Second, most implementations do not allow common debugging facilities like setting a breakpoint or stepping through the macro code. Third and most importantly, no type checking is done at all during macro expansion. The lack of type support prevents the C compiler from printing helpful diagnostic messages in many cases of problems and thus leads to code that is hard to maintain.

Modularisation of program fragments

So in the end, the decision to use the C Preprocessor for modularizing these fragments is based on technical circumstances. While we envision compositional language approaches for their technical handling like feature oriented programming (FOP) [1], aspect oriented programming (AOP) [4] or comparable languages, we should also consider the motivation for introducing these parts in form of function fragments in the first place. Linux is a very implementation driven project for mainly two reasons. As indicated before, operating systems are inherently implementation driven. Moreover, Linux is a high traffic free-software project with a very active development community. For these reasons, many optional features of various kinds have been proposed and integrated into the Linux code base.

However, there are many cases where two or more optional features behave differently when they are selected at the same time, compared to the case that only a single one is selected. This problem has already been discussed by Batory et. al as the *optional feature problem* [8, 11]. The proposed solution is to modularize the features as *derivatives*, so that the variant management system can select these *derivative modules* according to the needs of the implementation in a given configuration.

In Linux these *derivatives* are not modularized at all, but scattered across the Linux source base using the C Preprocessor. Because nothing tracks the consistency of these modules to the family models, *unused* derivatives become easily orphaned but still end up in the resulting product (the bootable Linux kernel image). This can result in modules that can be identified by preprocessor statements but can never be enabled or disabled. For this reason, we call these *undead* modules *zombies*.

5. RELATED WORK

Lotufo [12] analyzes the complexity of maintaining the Kconfig files. An investigation of 29 stable versions of the Linux kernel configuration options is presented. He concludes that the complexity of the code for the configuration options increases consistently, as well as the complexity of the resulting model. Interestingly, as we point out in this work, he also suggests that reasoning capabilities should be added to *Kconfig*.

Post and Sinz [16] present a technique called lifting that converts all variants of a SPL into a meta program in order to facilitate the application of verification techniques like static analysis, or model checking. They evaluate their approach by applying it into to the `Kconfig` files of the Linux kernel: The `Kconfig` files were converted into C-code for analysis with a source code checker, which reveals two new bugs attached to uncommon configuration. This fact also supports the idea that the Linux kernel should introduce *reasoning* capabilities for its variant model.

Kästner et. al. [7] present an approach to check the syntactic correctness of all variants of a software product line. They present the tool CIDE which is able to analyze `CPP`-based code among other languages. The concept of finding bugs introduced by the use of `CPP` directives is similar to our work, however, while CIDE focuses on syntactic errors, our approach finds inconsistencies between the source code and the variant model directly.

6. CONCLUSIONS

The mapping between the implementation of the variability points in the source code and the family model is incomplete and inaccurate. Our investigation of the Linux kernel shows that the mapping between the implementation of the variability points in the source code and the family model shows obvious inconsistencies. Our first probably inaccurate, but conservative checks indicate over 300 real bugs that arise from conditional compilation blocks which use configuration options that are never defined in the *Kconfig* variant model.

We believe that this number is only the tip of the iceberg and expect that additional bugs from defined *Kconfig* items that are never used in the source code remain undetected. Besides these violations of *referential integrity*, we also believe that more inconsistencies can be detected by checking for

semantic integrity, that is if the condition of a conditional block can be satisfied in the *Kconfig* variant model for all branches that is defined by the condition in the `#if` or `#ifdef` preprocessor statement.

It is clear that the impressive amount of variability cannot be checked by kernel developers manually in the source code. We therefore propose a tool that accompanies *Kconfig*, but represents the counterpart for kernel developers. This tool will help kernel developers to check all consistency conditions reliably. According to our estimations we are confident that our approach is feasible for the impressive amount of variability points in the Linux code base.

7. REFERENCES

[1] D. Batory, J. N. Sarvela, and A. Rauschmayer. Scaling step-wise refinement. In *Proceedings of the 25th International Conference on Software Engineering (ICSE '03)*, pages 187–197, Washington, DC, USA, 2003. IEEE Computer Society Press.

[2] D. Beuche, A. Guerrouat, H. Papajewski, W. Schröder-Preikschat, O. Spinczyk, and U. Spinczyk. The PURE family of object-oriented operating systems for deeply embedded systems. In *Proceedings of the 2nd IEEE International Symposium on Object-Oriented Real-Time Distributed Computing (ISORC '99)*, pages 45–53, St Malo, France, May 1999.

[3] D. Beyer. Relational programming with crocopat, 2006.

[4] K. Czarnecki and U. W. Eisenecker. *Generative Programming. Methods, Tools and Applications*. Addison-Wesley, May 2000.

[5] E. W. Dijkstra. The structure of the THE-multiprogramming system. *Communications of the ACM*, 11(5):341–346, May 1968.

[6] A. N. Habermann, L. Flon, and L. W. Cooprider. Modularization and hierarchy in a family of operating systems. *Communications of the ACM*, 19(5):266–272, 1976.

[7] C. Kästner, S. Apel, S. Trujillo, M. Kuhlemann, and D. Batory. Guaranteeing syntactic correctness for all product line variants: A language-independent approach. In *Proceedings of the 47th International Conference Objects, Models, Components, Patterns (TOOLS EUROPE)*, volume 33 of *Lecture Notes in Business Information Processing*, pages 175–194. Springer Berlin Heidelberg, June 2009.

[8] C. Kästner, S. Apel, S. S. ur Rahman, M. Rosenmüller, D. Batory, , and G. Saake. On the impact of the optional feature problem: Analysis and case studies. In *Proceedings of the 13th Software Product Line Conference (SPLC '09)*, Washington, DC, USA, 2009. IEEE Computer Society Press.

[9] B. W. Kernighan and D. M. Ritchie. *The C Programming Language*. Prentice Hall PTR, Englewood Cliffs, NJ, USA, 1978.

[10] Linus Torvalds. Sparse - a semantic parser for C. `http://www.kernel.org/pub/software/devel/sparse/`, 2003.

[11] J. Liu, D. Batory, and C. Lengauer. Feature oriented refactoring of legacy applications. In *Proceedings of the 28th International Conference on Software Engineering (ICSE '06)*, pages 112–121, New York, NY, USA, 2006. ACM Press.

[12] R. Lotufo. On the complexity of maintaining the linux kernel configuration. Technical report, Electrical and Computer Engineering University of Waterloo, 2009.

[13] A. Massa. *Embedded Software Development with eCos*. New Riders, 2002.

[14] M. Mendonça, A. Wasowski, K. Czarnecki, and D. D. Cowan. Efficient compilation techniques for large scale feature models. In *Proceedings of the 5th International Conference on Generative Programming and Component Engineering (GPCE '08)*, pages 13–22, 2008.

[15] D. L. Parnas. On the design and development of program families. *IEEE Transactions on Software Engineering*, SE-2(1):1–9, Mar. 1976.

[16] H. Post and C. Sinz. Configuration lifting: Verification meets software configuration. In *Proceedings of the 23th IEEE International Conference on Automated Software Engineering (ASE '08)*, pages 347–350, 2008.

[17] J. Sincero, H. Schirmeier, W. Schröder-Preikschat, and O. Spinczyk. Is the linux kernel a software product line? In F. van der Linden and B. Lundell, editors, *Proceedings of the International Workshop on Open Source Software and Product Lines (SPLC-OSSPL 2007)*, Kyoto, Japan, 2007.

[18] J. Sincero and W. Schröder-Preikschat. The linux kernel configurator as a feature modeling tool. In S. Thiel and K. Pohl, editors, *SPLC (2)*, pages 257–260. Lero Int. Science Centre, University of Limerick, Ireland, 2008.

[19] H. Spencer and G. Collyer. #ifdef considered harmful, or portability experience with C News. In *Proceedings of the 1992 USENIX Technical Conference*. USENIX Association, June 1992.

[20] M.-A. D. Storey, K. Wong, P. Fong, D. Hooper, K. Hopkins, and H. A. Müller. On designing an experiment to evaluate a reverse engineering tool. In *Proceedings of the 3rd Working Conference on Reverse Engineering, (WCRE'96), Monterey, California, USA, November 8-10, 1996*, pages 31–, Washington, DC, USA, November 1996. IEEE Computer Society Press.

Feature-Oriented Refinement of Models, Metamodels and Model Transformations

Salvador Trujillo Ander Zubizarreta Xabier Mendialdua Josune de Sosa
IKERLAN Research Centre, Spain
{strujillo, ander.zubizarreta, xmendialdua, jdesosa}@ikerlan.es

Abstract

Done well, the blend of Model Driven Development (MDD) and Software Product Lines (SPL) offers a promising approach, mixing abstraction from MDD and variability from SPL. Although Model Driven Product Lines have flourished recently, the focus so far has been mostly on how to cope with the variability of models. This focus on model variability has limited however the extension of variability to further artifacts apart from models such as metamodels and model transformations, that may cope with variability too in a product line setting. In this paper, we address the application of feature-oriented refinement to models, metamodels and model transformations. We illustrate our work with a case study of an embedded system.

Categories and Subject Descriptors D.2.2 [Software Engineering]: Design tools and techniques; D.2.13 [Software Engineering]: Reusable Software

General Terms Design, Modeling

Keywords models, metamodels, model transformations, refinements, AHEAD, XAK

1. Introduction

Modeling is essential to cope with the complexity of current software systems during their entire life cycle. Models allow developers to precisely capture and represent relevant aspects of a system from a given perspective and at an appropriate level of abstraction. Initially, modeling aimed at the representation of individual software systems where a collection of models typically describe their static structure, dynamic behavior, interaction with the user, and so on.

Model-Driven Development aims at the automation of repetitive code. The key to get this benefit is the use of abstraction, which enables to raise the abstraction level. Doing so, it is possible to focus on the domain details and separate from the implementation details. Specific benefits from MDD are productivity, reduced cost, portability, drops in time-to-market, and improved quality [9]. Overall, the main economic reason behind following such approach is the productivity gain achieved, which is reported by some studies [7, 11].

Researchers and practitioners have recently realized the necessity for modeling variability of software systems where software product line engineering poses major challenges on contemporary modeling techniques. A software product line is a set of software intensive systems that are tailored to a specific domain or market segment and that share a common set of features [5, 13]. A feature is an end-user visible behavior of software systems, and features are used to distinguish different software systems, a.k.a. variants, of a software product line [8].

In embedded systems, there is typically a family of control systems sharing commonality and distinguished by variability. The latter happens when one system needs to control different types of elements (e.g., exclusive subsystems from different providers or different target hardware platforms with distinct operative systems) implies that each needs to be controlled in a variable way. This has been often achieved by introducing the notion of features that are not necessarily present in all possible variants. This impacts not only on the implementation, but on the modeling level too.

So far, two main facets of modeling have been used in contemporary software product lines [2]. First, there are approaches for describing the variability of a software product line (e.g, there are feature models that specify which feature combinations produce valid variants [8]). Second, all variants in the product line may have models that describe their structure, behavior, etc. More to the point, such models may account for variability. Indeed, there is an increasing wealth of work on model variability where feature-oriented model refinement is just another approach [16].

Although when dealing with variability in an MDD scenario, there are further artifacts apart from models that need to cope with variability, the wealth of work on model variability has not a counterpart with other MDD artifacts. This paper takes a step back to study such impact into a broader perspective. We shift our attention from the variability of models to the variability of model transformations, that define the mappings between models and models and code. Since model transformations are typically defined at the metamodel level, we also analyze the impact of variability on metamodels. This perspective shift provides a bigger picture on the application of feature-oriented software development by also including metamodels and model transformations.

In this paper, we cope with the variability of models, metamodels and model transformations by applying the notion of step-wise refinement to them [3]. Thus, a realization of a feature consists of refinements of models, metamodels and model transformations. To illustrate our ideas, we introduced a simplified case study. We begin by reviewing the background.

2. Background

2.1 Model-Driven Development

Model Driven Development is a paradigm where models are used to develop software. This process is driven by model specifications and by transformations among models or models and code. It is

FOSD'09, October 6, 2009, Denver,Colorado, USA.

the ability to transform among different model representations that differentiates the use of models for sketching out a design from a more extensive model-driven software engineering process where models yield implementation artifacts. This paradigm eases cumbersome and repetitive tasks, and paves the way for productivity gains.

The main artifacts involved in MDD are models, metamodels and model transformations.

2.1.1 Models

Model is a term widely used in several fields with slightly different meanings. A model represents a part of the reality called the object system and is expressed in a modeling language. A model provides knowledge for a certain purpose that can be interpreted in terms of the object system [10]. Typical examples of models involve statecharts that represents the behavior of some functionality or class diagrams that show the structure of some code.

We illustrate our ideas with a family of embedded systems developed following MDD and SPL. RainCreek is a flooding system that controls a set of subsystems, typically with behavior, inputs, outputs, etc. There is a cooling subsystem that is open/closed based on data gathered from temperature sensors. Figure 1a shows a simplified model representing a part of such subsystem where inputs are defined based on a metamodel explained next. Figure 1b is the textual representation of Figure 1a where RainCreek `<System>` is defined as a set of `<System.subsystems>`. Cooling is one subsystem. Each `<Subsystem>` has a set of `<Subsystem.inputs>` defined by `<Input>`. The files used in this example are available online[1].

2.1.2 Metamodels

A model is frequently considered an instance conforming to a metamodel. A metamodel is a model of a modeling language where the language is specified [10]. In other words, the metamodel describes the elements of the domain and their relationships.

In our case study, the metamodel is defined using Ecore, showing the elements, relationships, data-types, operations and basic constraints that a model can have. Figure 2 shows a simplified metamodel for the model introduced earlier for RainCreek. Note that the model in Figure 1 conforms to this metamodel. This metamodel introduces not only the types of elements that can be used within the model, but also their hierarchical relationships. More important, the metamodel states also some constraints among those types. An Ecore class has been defined for each element type that the model can have. Containment relationships have been used to define the hierarchy of the metamodel elements. Every subsystem must have at least one input, while there is no maximum value.

The mappings between metamodels are typically represented by model transformations.

2.1.3 Model Transformations

Model transformations play a pivotal role in MDD because they turn the use of models for drawing into a more extensive model-driven usage where implementations can be directly obtained [14]. A model transformation is the process of converting one model to another model [12]. In general, a model transformation is the process of automatic generation from a source to a target model, according to a transformation definition, which is expressed in a model transformation language [10].

Depending on the target, model transformations can be model-to-model or model-to-text transformations. The former takes as input one or more models conforming to given metamodels and produces one or more models conforming to the same or other metamodels. The latter produces text as its output, that can be implementation code, documentation, or any other textual form.

Model-to-model transformations usually make use of rules that are defined as mappings between input and output metamodels. Model-to-text transformations combine rules with text templates that define the form of the output text. There is a variety of open-source and commercial tools and languages for model transformations such as ATL [4], RubyTL [6], ATC [15], and MOFScript (http://www.eclipse.org/gmt/mofscript/).

Our case study focuses on model-to-text transformations defined using MOFScript transformation language. MOFScript language is a metamodel-independent language that allows to use any kind of metamodel and its instances for text generation. MOFScript tool is based on EMF and it is available as an Eclipse plugin.

The definition of a MOFScript transformation consists of rules. Figure 3 shows a base transformation definition for transforming the RainCreek model shown earlier into C++ code. Implementation-wise, a transformation module called `rc2code` is defined. Such module is composed by several rules. They can have a context type scoping to which metamodel elements can such rules be applied.

Different rules have been defined to generate different parts of the output code. Each rule defines the appearance of part of the output text and introduces model elements in it. The main() rule defines the name of the output file and calls to all the other rules.

2.2 On the Refinement of Models

The refinement of models allows to extend a model adding elements, in order to enable their customization for different variability needs. This way when working with a family of products a base model with the common elements can be reused [16].

2.2.1 Base of a Model

The representation of models is typically done as instances of metamodels represented in Ecore. Hence, it is common to find models represented in textual form with an XML syntax.

Figure 1 shows our example `RainCreek` model, whose representation is XML. This model is an instance of the source metamodel defined with Ecore, and so can have elements defined in such metamodel (see Figure 2). The model shows a flooding system of `RainCreek` whose `Cooling` subsystem has two inputs: `Temp` and `Flow`. The data type for each input is defined with the `Input.type` element.

In a product-line setting, a model may demand to be extended, so model refinement is used to add new elements[16].

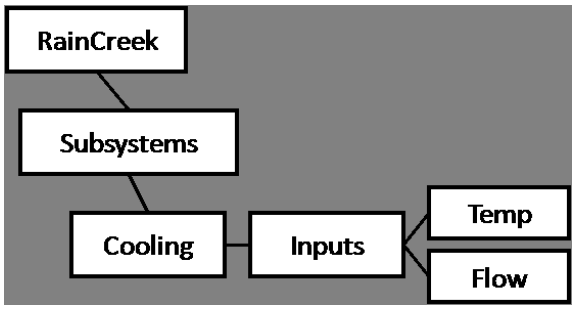
2.2.2 Refinement of a Model

In general, a refinement can be deemed of as a function that takes an artifact as an input, and returns another similar artifact which has been leveraged to support a given feature [3]. The refinement of models is done addressing the units of refinement in the base model, while deltas are defined as refinements.

XAK is a language for defining refinements in XML documents and it is accompanied by a validator and a composer tool [1]. Using XAK is possible to specify which elements in a base XML document can be refined and also to define the refinements. We used XAK for the refinement and composition of models.

Any traditional XML document can be a base document, but some additional annotations are needed (see Figure 1). The attributes `@xak:artifact` and `@xak:feature` are added to the document element (i.e. the root). The first one specifies the name of the document that is being incrementally defined, while the second one specifies the name of the feature being supported (“base” for base documents). To specify which elements are modularizable, `@xak:module` is used. That is, it indicates those elements that play

[1] http://www.ikerlan.es/softwareproductline/fosd2009examples.zip

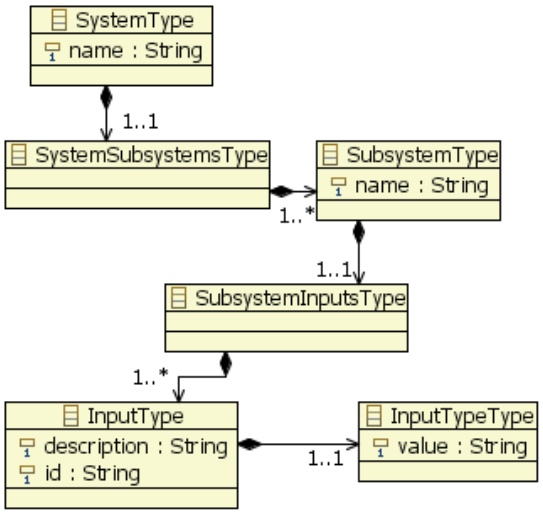


(a)

```
1  <System name="RainCreek" xak:artifact="baseModel.rc"
       xak:feature="base">
2    <System.subsystems>
3      <Subsystem xak:module="mCooling" name="Cooling">
4        <Subsystem.inputs>
5          <Input xak:module="mTemp" id="Temp"
               description="Temp ...">
6            <Input.type value="int"/>
7          </Input>
8          <Input xak:module="mFlow" id="Flow"
               description="Flow ...">
9            <Input.type value="int"/>
10         </Input>
11       </Subsystem.inputs>
12     </Subsystem>
13   </System.subsystems>
14 </System>
```

(b)

Figure 1. A RainCreek Model (simplified)



(a)

```
1  <ecore:EPackage xak:module="mPackage" ...>
2  <!-- Content omitted -->
3    <eClassifiers xak:module="mSubsys" xsi:type="
         ecore:EClass" name="SubsystemType">
4      <eStructuralFeatures xsi:type="ecore:EReference"
           name="subsystemInputs" lowerBound="1" eType="
           #//SubsystemInputsType" containment="true"
           resolveProxies="false">
5        <!-- Content omitted -->
6      </eStructuralFeatures>
7      <eStructuralFeatures xsi:type="ecore:EAttribute"
           name="name" lowerBound="1" eType="
           ecore:EDataType␣http://www.eclipse.org/emf
           /2003/XMLType#//String">
8        <!-- Content omitted -->
9      </eStructuralFeatures>
10   </eClassifiers>
11   <!-- Content omitted -->
12 </ecore:EPackage>
```

(b)

Figure 2. RainCreek Metamodel

the role of modules, and henceforth can be refined. In general, a XAK module has a unique name.

Refinement documents refine base documents and they have `<xak:refines>` element as root (see Figure 4). Its content describes a set of module refinements (i.e. elements annotated with the `xak:module` attribute) over a given base document (i.e. the `xak:artifact` attribute). The `xak:super` node is a marker that indicates the place where the original module body is to be substituted. In general, a XAK refinement document can contain any number of `xak:module` refinements.

Those elements of the model that can be refined must be defined before refining such model. In our example the refinement units are the `<Subsystem>` element and `<Input>` elements. That is, we can add new elements to the subsystem, or we can add new elements to the inputs. (Figure 1 showed the base model.) A base is designated as an `xak:artifact` to denote its ability to be refined. Each artifact may expose its refinable parts beforehand. We add `xak:module` attribute to `<Subsystem>` and all `<Input>` elements to specify that they are refinables. Note that not all elements are refinable, but only those we annotate beforehand by `xak:module`.

Many refinements can be applied to our base `RainCreek` model. For example the `Cooling` subsystem of our model could have parameters, in addition to inputs. Alternatively, the input size could be defined apart from its data type, to make our model compatible with a specific platform.

To give an example, we show how we can add parameters to a subsystem. Figure 4 shows the refinement that has been defined to add two parameters to the `Cooling` subsystem. This refinement refines the base model adding parameters defined with a `ParamType` element, within a `Subsystem.params` element, similarly to the definition of inputs. We define the base model that is to be refined using `xak:artifact="baseModel.rc"`. The use of `<xak:extends xak:module="mCooling">` designates the module that is to be refined. The use of `<xak:super xak:module="mCooling"/>` indicates that the body of the original `mCooling` module will be placed there (i.e., a form of reuse).

```
texttransformation Rc2code (in mdl: ... ) {
  mdl.SubsystemType::main() {
  file (self.name + ".cpp")
    /* Content omitted */
    self.generateStart ()
    self.generateExec ()
  }
  mdl.SubsystemType::generateStart() {
  '
/* Content omitted */
bool 'self.name'Impl::start() {
  // Initialization of the variables //
  'self.initAll()'
}
  '
  }
  mdl.SubsystemType::initAll() {
    self.initInputs()
  }
  mdl.SubsystemType::initInputs() {
  '
  /*inputs*/
    'self.subsystemInputs.input->forEach (input:mdl.
        InputType) {
  'in'input.id' =  new Input ("'input.id'", "'
        input.description'") ;
  '
    }
  }
  /* Content omitted */
}
```

(a)

```
<MOFScriptModel:MOFScriptSpecification xak:artifact="
    base.m2t.model.mofscript" xak:feature="base" ...>
  <transformation xak:module="mRc2Code" name="Rc2code
      " ...>
    <parameters name="mdl" type= .../>
    <transformationrules isEntryPoint="true" name="
        main" ...>
      <statements xsi:type="
          MOFScriptModel:FileStatement" ...>
      <!-- Content omitted -->
    </transformationrules>
    <!-- Content omitted -->
    <transformationrules xak:module="mInitAll" name="
        initAll">
      <statements xsi:type="
          MOFScriptModel:FunctionCallStatement">
        <function name="self.initInputs" .../>
      </statements>
      <!-- Content omitted -->
    </transformationrules>
    <transformationrules name="initInputs">
      <!-- Content omitted -->
      <statements xak:module="mInputIt" type="mdl.
          InputType" variable="input" ...>
        <!-- Content omitted -->
        <source  name="self.subsystemInputs.input"
            .../>
      </statements>
    </transformationrules>
  </transformation>
</MOFScriptModel:MOFScriptSpecification>
```

(b)

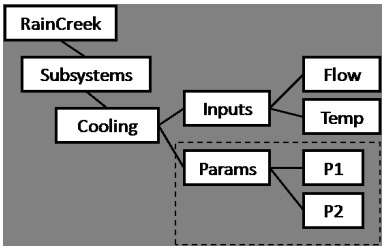
Figure 3. Example of a Model Transformation

```
<xak:refines xak:artifact="base.rc"
    xak:feature="paramDelta" ...>
  <xak:extends xak:module="mCooling"
      >
    <xak:super xak:module="mCooling"
        />
    <Subsystem.params>
      <Param id="P1" description="
          Param for ...">
        <Param.type value="int"/>
      </Param>
      <Param id="P2" description="
          Param for ...">
        <Param.type value="int"/>
      </Param>
    </Subsystem.params>
  </xak:extends>
</xak:refines>
```

(a)

```
<System name="RainCreek"  ...>
  <System.subsystems>
    <Subsystem ...>
      <Subsystem.inputs>
        <!-- Content omitted >
      </Subsystem.inputs>
      <Subsystem.params>
        <Param description="Param
            for ..." id="P1">
          <Param.type value="int"/>
        </Param>
        <Param description="Param
            for ..." id="P2">
          <Param.type value="int"/>
        </Param>
      </Subsystem.params>
    </Subsystem>
  </System.subsystems>
</System>
```

(b)



(c)

Figure 4. Composition of models: (a) Refinement; (b) Composed model (textual); (c) Composed model

Finally, the new elements to be introduced are defined after the `<xak:super>` element.

A distinguishing characteristic of our approach is that a separated and independent refinement is defined for each additional feature we would like to add to the base model. To synthesize a resulting model, the base model and the refinements are composed.

2.2.3 Composition of Base and Refinement Models

The composition of the base model and the refinement is achieved using XAK composer tool [1]. In our example, to extend the RainCreek base with the refinement model, the following composition is needed:

> *xak -xak2 -c baseModel.rc paramModelDelta.xak -o compParamModel.rc*

The execution of the above command invokes the XAK composer tool[2].

Figure 4b shows the resulting model, which results from the synthesis of a base model with refinements (Figure 1b and Figure 4a, respectively). Note that in this model a new `<Subsystem.params>`, `<Param>` and `<Param.type>` are defined. This refines the `Cooling` subsystem with `P1` and `P2` parameters. Although our example is restricted to the composition of a model refinement with a base model, typically more than one model refinements can be composed.

However, the resulting composed model does not conform to the metamodel introduced earlier. This is because param-related elements were not defined in the metamodel. This conformance relationship between the model and the metamodel implies that the metamodel may need to be refined together with the model, which we call as *Lock-Step Refinement*.

3. Beyond Model Refinement

This section elaborates on the refinement of metamodels and model transformations.

3.1 Scenario

A Model-Driven Development scenario for individual programs typically involves artifacts such as models, metamodels and model transformations. In a software product line setting with a family of programs, there is a need for their customization. Thus, we elaborate on the need for the refinement of models, metamodels, and model transformations.

There are different scenarios when using models in software product lines. Consider the differences on the modeling language used: it is not the same to use UML or a Domain Specific Language (DSL).

There are scenarios where model variability may be enough and the variability of metamodels or model transformations may not be needed. This may happen in situations where the used metamodel is standard and so it is not subject to variability. For instance, when using a UML class diagram, it seems unlikely to make its metamodel variable since it is somehow standardized. This may apply generally to the metamodels within the UML. A similar situation occurs when the model transformations come from a common library of model transformations that are shared. As a case in point, consider the dozens of model transformations expressed in ATL that are available online[3].

[2] The detail of the command is as follows. `xak` is the executable, `-xak2` is an option to use xak2 syntax of XAK, `-c` option introduces the artifacts to be composed (first base, and so forth), `-o` option specifies the synthesized result.

[3] http://www.eclipse.org/m2m/atl/atlTransformations/

Therefore, in scenarios with standard metamodels and/or shared transformations, the use of model variability seems enough and thus variability of model transformations may not be needed. However, there exist other scenarios where, in addition to the variability of models, the variability of metamodels and model transformations may be needed as well.

Consider the case in which different models, metamodels and model transformations may need to be customized for different targets. Model transformations share a significant common part while differing in some variable parts. In our case study, different implementation code shall be generated from the same source model. The target code is expected to be executed in different platforms (e.g. Windows CE or VxWorks). This situation could be well handled by defining features of the software product line that impact on the model transformation. There is a large proportion of shared code and some particularities are bound to each programming language. For instance, the way real-time code is implemented differs among platforms, while keeping a significant common part.

In such situation, the application of variability to model transformations may enable to centralize those differences in a unique model transformation. This may impact as well on the metamodel, since model transformations are typically mappings between metamodels. In the case for different platforms explained above, there is a need to cope with metamodel variability as well. Instead of different metamodel targets from different model transformations, there is a model transformation with variability and so its associated metamodels may cope with such variability. Note that this variability in time is not the variability in space imposed by the metamodel's evolution.

Our motivating case study is an scenario that demands to cope with such variability.

3.2 Overview

The artifacts involved in a Model-Driven Development scenario are related each other. Models conform to metamodels. Model transformations define the mappings between metamodels. The execution of a model transformation produces an output model from an input model (or a set of models). Those models conform to the source and target metamodel, respectively. Indeed, the relationship among those elements is well-known.

The notion of *Lock-Step Refinement* does not refer to the common understanding of that relationship explained above. Conversely, it refers to the relationships among increments or refinements of those elements.

Consider a model refinement Δm_1 as an extension of model m_1. The model m_1 conforms to the metamodel MM_A. In general, this mechanism realizes the variability of a feature where Δm_1 is a part of. There appears two situations depending whether the content of Δm_1 conforms to MM_A or not. When Δm_1 conforms to the existing metamodel MM_A, there is no need to trigger a refinement of such metamodel. Conversely, if Δm_1 introduces newer content that does not conform to the existing metamodel, there is a need to refine the metamodel MM_A with a metamodel refinement ΔMM_A.

In the situation where a model refinement triggers a metamodel refinement, we introduce the notion of lock-step refinement to capture such relationship. More to the point, when the model Δm_1 is refined, it depends on a refinement of its metamodel ΔMM_A. Doing so, the compound model may conform to the compound metamodel.

The lock-step relationship explained between a model and a metamodel also occurs between a metamodel and a model transformation. More generally, it may include additionally a refinement of model transformations such as $\Delta MT_{1\rightarrow 2}$ (a mapping between Δm_1 and Δm_2).

A Lock-Step Refinement is defined in this work as a set encompassing the refinements of models, metamodels and model transformations. For instance, a lock-step refinement ΔLSR_I may consist of Δm_1, Δm_2, ΔMM_A, ΔMM_B and $\Delta MT_{1\rightarrow 2}$.

Next, the refinement of metamodels and model transformations is described.

3.3 On the Refinement of Metamodels

In some of the scenarios presented above, there is a need to refine metamodels. To attain this, we followed an approach similar to the introduced earlier for models.

3.3.1 Base of a Metamodel

Metamodels are defined as instances of meta-metamodels. Ecore is the meta-metamodel used for representing our case study.

Figure 2 shows a simplified Ecore metamodel used in RainCreek. Ecore metamodels are serialized using *XML Metadata Interchange* (XMI). The metamodel is defined under the `<ecore:EPackage>` element. Classes and data-types are defined using `<eClassifiers>` element, while relationships and attributes are defined with the nested `<eStructuralFeatures>` element.

Figure 2b shows part of the XMI representation of the Ecore metamodel (Figure 2a), where the inputs/outputs of RainCreek and their relationships are defined. Such XML-based representation enables XAK to be used for the refinement.

3.3.2 Refinement of an Ecore-based Metamodel

The refinement of a metamodel may enable to extend it adding new element types, relationships, etc.

Similarly to model refinements, the first is to decide which of the elements in our base metamodel can be refined. By adding the `xak:module` attribute to an element, it is defined those elements that are modularizables. In our case, we designate `EPackage` and `EClass` as the modularizable elements. This allows us to add new classes to our metamodel or to extend the existing ones. Figure 2b shows how the `xak:module` attribute has been added to such elements.

Hence, for our metamodel to conform to the model shown earlier, a refinement to add the definition of `Subsystem.params`, `Param`, and `Param.type` elements to the metamodel is needed. `SubsystemParamsType`, `ParamType` and `ParamTypeType` classes and their relationships are defined in a refinement, in order to be added to the `EPackage` element. A new relationship between `SubsystemType` class and `SubsystemParamsType` is added to the `SubsystemType` class, already annotated with `xak:module=''mSubsys''` attribute. Figure 5a shows part of such refinements for the base metamodel.

To obtain the metamodel where the `Subsystem` has parameters apart from inputs, base metamodel and the refinement shown in Figure 5a need to be composed.

3.3.3 Composition of Base and Refinement Metamodels

Similarly to the composition of models, the base and the refinement metamodel definitions can be composed using XAK composer tool:

> *xak -xak2 -c baseRC.ecore paramMMDelta.xak -o compParamMM.ecore*

The output of the above command is the `compParamMM.ecore` metamodel that is shown in Figure 5b. The graphical representation of the composed Ecore metamodel is shown in Figure 5c. This metamodel defines the elements and relationships that a model may conform to. Since the metamodel has been refined adding classes and relationships to define parameters, the refined model now conforms to the refined metamodel. That is, the model shown in Figure 4 is now an instance of the metamodel shown in Figure 5.

3.4 On the Refinement of Model Transformations

A model transformation defines typically the mappings between metamodels. When a metamodel is refined to account for some feature as shown earlier, the mappings of the model transformation may be needed to be refined too for that feature.

3.4.1 Base of a Model Transformation

A base transformation definition has been defined with MOFScript (see Figure 3a). This definition takes a `RainCreek` model as input, and has the rules to generate C++ code as output.

In our example, the base transformation defines some rules to generate different parts of the output code, such as declaration of variables, implementation of the constructors and destructors, initializations, etc. These rules combine text with the elements defined in the metamodel.

MOFScript allows to represent a transformation definition as a model, conforming to the MOFScript metamodel. Figure 3b shows the model representation of the transformation using XMI. (Note that this figure is equivalent to the textual representation of the transformation in Figure 3a). The transformation module is defined with the `<transformation>` element. Nested to this element, variables, parameters and rules are defined using the `<variables>`, `<parameters>` and `<transformationrules>` elements, respectively.

This base transformation can be executed, giving as input the base model shown in Figure 1. The application of this transformation gives as output a class definition for the `Cooling` subsystem. However, if the input model is the refined one, the declaration and initialization of params will not be handled by the output code. This is because the model transformation does not consider the handling or even the existence of params. Thus, the transformation needs to be refined to generate the declaration and initialization of params in the output code.

3.4.2 Refinement of a MOFScript-based Transformation

Having the XMI representation of the transformation definition, it is possible to achieve its refinement using XAK.

The model of our transformation shown in 3b is the base transformation which needs to be refined. Those elements of the transformation that can be refined must be defined before refining such transformation. In our example, the refinement units are the transformation definition and their rules. That is, we can add new rules to the transformation definition or we can extend existing rules with new statements. We add `xak:module` attribute to `<transformation>` and `<transformationrules>` elements to specify that they are modularizables (see Figure 3b). In our example the rules that need to be refined are `generateLocals()` and `initAll()`, so it is possible to add the `xak:module` attribute only to their definitions. If we add `xak:module` to all `<transformationrules>` elements, all the rules can be refined.

Figure 6a shows a refinement for the base transformation. It refines the base transformation definition to introduce the declaration and initialization of params in the generated C++ code. The rules `generateLocals()` and `initAll()` are extended and a new rule called `initParams()` is added. The rule `generateLocals()` is extended to print the declaration of params after the declaration of inputs. `The initAll()` rule is extended to call the newly added `initParams()` rule, which prints the initializations of params in the output file.

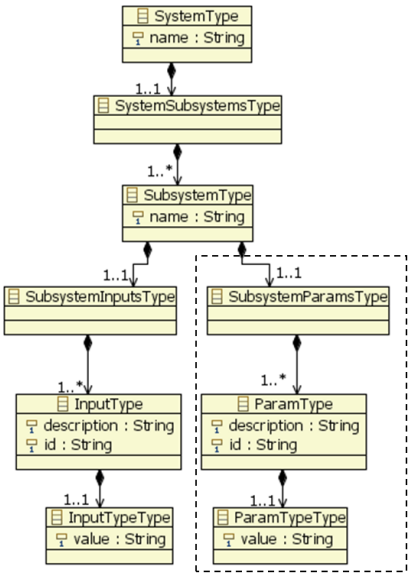
In order to get a transformation definition dealing with params, this refinement needs to be composed with the base transformation definition.

```
<xak:refines xak:artifact="baseRC.ecore" xak:feature="paramDelta" ...>
  <xak:extends xak:module="mSubsys">
    <xak:super xak:module="mSubsys"/>
    <eStructuralFeatures xsi:type="ecore:EReference" eType="#//SubsystemParamsType" ...>
      <!-- Content omitted -->
    </eStructuralFeatures>
  </xak:extends>
  <xak:extends xak:module="mPackage">
    <xak:super xak:module="mPackage"/>
    <!-- Content omitted -->
    <eClassifiers xsi:type="ecore:EClass" name="ParamType">
      <!-- Content omitted -->
    </eClassifiers>
    <eClassifiers xsi:type="ecore:EClass" name="ParamTypeType">
      <!-- Content omitted -->
    </eClassifiers>
  </xak:extends>
</xak:refines>
```

(a)

```
<ecore:EPackage name="RC" ...>
  <!-- Content omitted -->
  <eClassifiers name="SubsystemType" ...>
    <!-- Content omitted -->
    <eStructuralFeatures name="subsystemParams" ...>
      <!-- Content omitted -->
    </eStructuralFeatures>
  </eClassifiers>
  <eClassifiers name="SubsystemParamsType" ...>
    <!-- Content omitted -->
    <eStructuralFeatures name="param" ...>
      <!-- Content omitted -->
    </eStructuralFeatures>
  </eClassifiers>
  <eClassifiers name="ParamType" ...>
    <!-- Content omitted -->
  </eClassifiers>
  <eClassifiers name="ParamTypeType" ...>
    <!-- Content omitted -->
  </eClassifiers>
</ecore:EPackage>
```

(b)



(c)

Figure 5. Composition of metamodels: (a) Refinement; (b) Composed metamodel (XMI); (c) Composed metamodel (Ecore diagram)

```
<xak:refines xak:artifact="base.m2t.model.mofscript" xak:feature="paramDelta" ...>
  <xak:extends xak:module="mInitAll">
    <xak:super xak:module="mInitAll"/>
    <statements xsi:type="FunctionCallStatement">
      <function name="self.initParams" .../>
    </statements>
    <blocks .../>
  </xak:extends>
  <xak:extends xak:module="mRc2Code">
    <xak:super xak:module="mRc2Code"/>
    <transformationrules name="initParams">
      <!-- Content omitted -->
    </transformationrules>
  </xak:extends>
  <!-- Content omitted -->
</xak:refines>
```

(a)

```
<MOFScriptSpecification ...>
  <transformation ...>
    <!-- Content omitted -->
    <transformationrules name="initAll" ...>
      <!-- Content omitted -->
      <statements xsi:type="FunctionCallStatement">
        <function name="self.initParams" .../>
      </statements>
    </transformationrules>
    <!-- Content omitted -->
    <transformationrules name="initParams">
      <!-- Content omitted -->
    </transformationrules>
  </transformation>
</MOFScriptSpecification>
```

(b)

```
/* Content omitted */
mdl.SubsystemType::initAll() {
    self.initInputs()
    self.initParams()
}
/* Content omitted */
mdl.SubsystemType::initParams() {
"/*params*/\n    "
  self.subsystemParams.param->forEach(param: mdl.ParamType){
  "p"
param.id
' =  new Param ("'
param.id
'", "'
param.description + '");\n    '
  }
}
```

(c)

Figure 6. Composition of transformations: (a) Refinement; (b) Composed transformation (model); (c) Composed transformation (MOFScript)

3.4.3 Composition of Base and Refinement Transformations

Again, using XAK composer tool a new transformation definition can be synthesized by composing the base transformation definition and the refinement.

> xak -xak2 -c base.m2t.model.mofscript paramTransDelta.xak -o compParam.mofscript

The result of the composition is a composed transformation model (Figure 6b), which can be executed directly giving as input the refined `RainCreek` model or can be converted to a MOFScript transformation definition file (Figure 6c) using that option in MOFScript tool, and then executed. The output of the transformation is the `Cooling.cpp` file which defines the implementation of the methods for the `Cooling` class.

The use of step-wise refinement enables the refinement of models, metamodels and model transformations. The nature of the refinement may trigger lock-step refinements among models, metamodels and model transformations.

4. Conclusions

This paper presented an approach to apply step-wise refinement beyond models. Specifically, we focused on the refinement of metamodels and model transformations in a model-driven product line scenario. We observed lock-step relationships among some of those refinements that are closely inter-related by features. We believe that this work provides a step forward in the field of Feature-Oriented Software Development by pushing variability beyond models and embracing metamodels and model transformations into the field.

The contribution of this paper is to provide a broader perspective on the application of FOSD into MDD by extending the refinement to further artifacts apart of models. We applied our ideas with a case study of an embedded system.

In our case, we observed lock-step relationships among different elements. In the future, we plan to explore them in further detail. Specifically, we aim at analyzing the conformance relationships among a model refinement and its lock-step metamodel refinement. This conformance may impact on the transformations ultimately.

Acknowledgments

This work was co-supported by the Spanish Ministry of Science & Innovation under contract TIN2008-06507-C02-02. We thank Maider Azanza for her comments on earlier versions of this draft.

References

[1] F. I. Anfurrutia, O. Díaz, and S. Trujillo. On Refining XML Artifacts. In *ICWE*, pages 473–478, 2007.

[2] S. Apel, F. Janda, S. Trujillo, and C. Kaestner. Model Superimposition in Software Product Lines. In *2nd International Conference on Model Transformations (ICMT 2009), Zurich, Switzerland, June*, 2009.

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

[4] J. Bézivin, G. Dupe, F. Jouault, G. Pitette, and J. E. Rougui. First Experiments with the ATL Model Transformation Language: Transforming XSLT into XQuery. In *2nd OOPSLA Workshop on Generative Techniques in the context of MDA, Anaheim, California, USA, Oct 27*, 2003.

[5] P. Clements and L.M. Northrop. *Software Product Lines - Practices and Patterns*. Addison-Wesley, 2001.

[6] J. Sánchez Cuadrado, J. García Molina, and M. Menárguez Tortosa. RubyTL: A Practical, Extensible Transformation Language. In *2nd European Conference on Model Driven Architecture - Foundations and Applications (ECMDA-FA 2006), Bilbao, Spain, Jul 10-13*, pages 158–172, 2006.

[7] D. Herst and E. Roman. Model Driven Development for J2EE Utilizing a Model Driven Architecture (MDA) - Approach: A Productivity Analysis. Technical report, TMC Research Report, 2003.

[8] K. C. Kang and et al. Feature Oriented Domain Analysis Feasability Study. Technical Report CMU/SEI-90-TR-21, Software Engineering Institute, November 1990.

[9] Mikko Kontio. Architectural Manifesto: The MDA Adoption Manual. http://www-128.ibm.com/developerworks/wireless/library/wi-arch17.html.

[10] I. Kurtev. *Adaptability of Model Transformations*. PhD thesis, University of Twente, 2005.

[11] OMG. MDA Success Stories. http://www.omg.org/mda/products_success.htm.

[12] OMG. MDA Guide version 1.0.1. OMG document 2003-06-01, 2003.

[13] K. Pohl, G. Bockle, and F. van der Linden. *Software Product Line Engineering - Foundations, Principles and Techniques*. Springer, 2006.

[14] S. Sendall and W. Kozaczynski. Model Transformation: The Heart and Soul of Model-Driven Software Development. *IEEE Software*, 20(5):42–45, 2003.

[15] A. Sánchez-Barbudo, E. V. Sánchez, V. Roldán, A. Estévez, and J.L. Roda. Providing an Open Virtual-Machine-based QVT Implementation. In *Proceedings of the V Workshop on Model-Driven Software Development. MDA and Applications (DSDM'08 - XIII JISBD)*, 2008.

[16] S. Trujillo, D. Batory, and O. Díaz. Feature Oriented Model Driven Development: A Case Study for Portlets. In *29th International Conference on Software Engineering (ICSE 2007), Minneapolis, MN, USA, May*, 2007.

Model-Driven Development of Families of Service-Oriented Architectures

Mohsen Asadi[1], Bardia Mohabbati[1] , Nima Kaviani[2], Dragan Gašević[3], Marko Bošković[3], Marek Hatala[1]
[1]*Simon Fraser University,* [2]*University of British Columbia,* [3]*Athabasca University, Canada*
{masadi,mohabbati,mhatala}@sfu.ca, nimak@ece.ubc.ca, dgaseavic@acm.org, marko.boskovic@athabascau.ca

ABSTRACT

The paradigms of Service Oriented Architecture (SOA) and Software Product Line Engineering (SPLE) facilitate the development of families of software-intensive products. Software Product Line practices can be leveraged to support the development of service-oriented applications to promote the reusability of assets throughout the iterative and incremental development of software product families. Such an approach enables various service oriented business processes and software products of the same family to be systematically created and integrated. In this paper, we advocate integration of software product line engineering with model driven engineering to enable a model driven specification of software services, capable of creating software products from a family of software services. Using the proposed method, we aim to provide a consistent view of a composed software system from a higher business administration perspective to lower levels of service implementation and deployment. We demonstrate how Model Driven Engineering (MDE) can help with injecting the set of required commonalities and variabilities of a software product from a high level business process design to the lower levels of service use.

Categories and Subject Descriptors

D.2.10 [Design]: Methodologies, D.2.13 [Reusable Software]: *Domain Engineering*, K.6.1 [Project and People Management]: Systems development

General Terms

Management, Design, Languages

Keywords

Software Product Lines, Business Process Management, Service-Oriented Architectures, Semantic Web.

1. Introduction

The growing opportunities for automated collaboration and communication offered by Internet technologies and the World Wide Web force a change in software development from self-contained and isolated software development towards enterprise level collaborative exchange of services and components. Such a change in software system use mandates the invention of methods, tools, and technologies for providing efficient and flexible ways for integration of distributed software services. Service Oriented Architecture (SOA) is appearing as the most promising paradigm for distributed computing application design, development, and deployment. In SOAs, software services are regarded as *autonomous, platform-independent, computational elements that can be described, published, discovered, orchestrated, and programmed* using standard protocols for building interoperating applications [1]. Success stories from industry (e.g. Amazon and Google) show high acceptance potentials in using SOA for distributed system development.

An ideal solution for addressing the newly emerged requirements and needs for current integration of distributed services in a SOA architecture demands for a coherent development process across various levels of development and integration. Achieving this cohesion starts by defining business processes and by specifying the set of coordinated activities realizing the goals of the business processes. Then, the software elements required to achieve these goals (i.e., the unit services) are identified, and for each unit service, the set of service interfaces realizing the specification of the unit service, as well as their corresponding service implementations, get identified and developed [2]. Going down the stages of the development lifecycle for SOA, there might be alternative choices on the set of business activities to accomplish business processes; there might be several unit services offering identical functionalities to business activities; and there may exist a diverse set of service interfaces and their corresponding implementations falling under a unit service. The noticeable variability in selecting business activities, unit services, service interfaces, and their implementations, opens room for employing Software Product Line Engineering (SPLE) as a methodological approach to address variability in these different layers of SOA development. A Software Product Line represents a set of software intensive systems that share a common managed set of features satisfying the specific needs of a particular market segment [3]. Applying SPLE to SOA development helps with providing a proper conceptualization of how alternatives for business processes and services in a distributed environment can be managed in order to achieve a valid software product at the end of the process. Bringing the benefits of Model Driven Engineering (MDE) into the process, we can abstract away the low-level technical specificities of development and pay more attention to a systematic development of business activities, their corresponding unit services, and service interfaces and components that materialize the unit services.

In this paper, we advocate a methodology combining the SPLE with MDE in order to bridge the gap between business process management and software engineering. Chang and Kim described layered variability modeling in SOA by providing a taxonomy of variability types which are derived from four layers specified as Business Processes, Unit Services, Service Interfaces, and Service Implementations [2]. We demonstrate how domain engineering can be consistently conducted across all four layers mentioned above, in order to decide about the families of SOA from their business process specification at the topmost layer to their concrete implementation at the lowest layer. We also show how employing model driven engineering principles enables the high level requirements at the business process layer to be combined with the technical requirements of software engineers to provide

implementation of service interfaces for the already defined business activities and unit services.

The rest of the paper is organized as follows. Section 2 provides background and concepts definition. Method overview is described in section 3. Detailed definition of artifacts created by methods is presented in section 4. Section 5 includes related works. Conclusion remarks and future works finally discussed in Section 6.

2. Background

The Software Product Line Engineering (SPLE) discipline provides methods for managing variabilities and commonalities of core software assets in order to facilitate the development of families of software-intensive products. The methodology presented in this paper serves for development of families of SOAs for implementing business processes. Feature modeling as one distinguished technique to model variability is employed for capturing variabilities in business processes and supporting software systems. For this reason, two additional underpinnings of this methodology, i.e., Model-Driven Engineering and Business Process Modeling, are explained in the next two subsections.

2.1 Model-Driven Engineering

Model-Driven Engineering (MDE) is a software engineering approach in software development which moves the focus of software development from implementation to the problem domain by raising the abstraction in software artifact development and automating implementation with means of transformation.. One of the realizations of MDE is the Model Driven Architecture (MDA) introduced by the Object Management Group (OMG). The MDA, additionally to abstraction and automation suggests the reliance on OMG standards like MOF, XMI, and UML in order to improve reusability and portability of designed software artifacts [25]. The abstraction is in MDA managed inin three layers::system requirements are specified in the *Computation-Independent Model* (CIM); the *Platform Independent Model* (PIM) abstracts away concerns regarding the implementation platforms from the system design; finally the *Platform Specific Model* (PSM) augments the PIM with platform specific concerns. The final code is generated from a PSM which corresponds to a specific implementation platform of choice. Producing PIM from CIM and PSM from PIM, as the heart of MDA, enables the concepts of each of the upper layers to be realized in the lower layers, bringing the model closer to the desired final deployment model. Nowadays, many model transformation approaches have been developed which cope with problems in this domain, (e.g., ATL[4]).

2.2 Business Process Modeling

According to [5], "a business process consists of a set of activities that are performed in coordination in an organizational and technical environment. These activities jointly realize business goals. Each business process is enacted by a single organization, but it may interact with business processes performed by other organizations". Business process management is defined as a set of management activities, and techniques to design, implement, evaluate, and execute business processes [5][6]. Main parts of business process management are the lifecycle and the business process modeling language (BPML). Various types of lifecycles have been defined to elaborate on steps and tasks which should be performed to design, implement, evaluate and enact business processes [6]. This diversity also applies to BPMLs, where there have been many different proposals [23][24].

BPMLs are used to represent artifacts produced by performing activities of lifecycle. Some of these modeling languages such as UML2.0 Activity Diagram, Business Process Definition Meta-model, Business Process Notation, etc. were evaluated with a framework provided by List & Korhert [7]. Business Process Modeling Notation (BPMN) is One of the prominent BPMLs which is used in the design phase of a lifecycle. BPMN represents processes as activities where the primary concern is the flow of control. However, recent research also proposes specializations of BPMN to model also business processes from the perspective of interactions of collaborating parties (e.g., iBPMN). Regarding selected platforms for running business processes in the implementation phase of a lifecycle, such as SOA or workflow management systems, other languages are used. For instance Business Process Executing Language (BPEL) is used for implementation and execution of business processes through composition (i.e., orchestration) of software services.

Nowadays, most of business process management approaches emphasize on the development of a single business process. There are many variants of the same business process, which are specialized according to various organizational needs. Business processes can have some parts common for a group (i.e., family) of different application cases, where involves some variability in selecting activities of a business process. A common functionality can be developed as a reusable asset to be utilized for creating new variants of business processes in a domain. Recognizing these phenomena and connecting the business process modeling problems with the principles of SPLs, a new term was coined: Business Process Family Engineering (BPFE) [19]. BPFE applies concepts of SPLE in business process management to the development of families of business processes. BPFE uses the same dual-lifecycle development as software product line engineering, i.e., domain engineering and application engineering. Therefore, the first lifecycle develops families of business processes by applying modified lifecycle of business process management which can identify common and variable features in a family and create reusable assets and reference models. Then, application engineering creates specific business processes for the target organization. Similar to the application engineering phase in Software Product Lines Engineering, here, application engineering is performed by choosing the desired features of the feature model.

2.3 Sample Case Scenario

To elucidate the proposed method and its activities in next sections, we use a "Supply Chain Management Application" (SCMA) exemplified as follows: We assume that in a typical shopping scenario, a customer pays the price of goods by one of the following payment methods: debit card, credit card, cash, or cheque. Then purchased goods get delivered to the customer. Every business process in our scenario presumably should support at least two methods of payment with one payment method always selected as cash payment. Some business processes support fraud detection for more security. For the process of delivery, first quality of goods should be verified, and then goods are packed and delivered to the customer by one of the delivery methods selected according to some criteria such as cost of delivery, time to arrive, etc.

As the paper proceeds, in order to explain outcomes of this method's activities, different models specifying SCMA business process family. and produced in different activities of this method, are demonstrated.

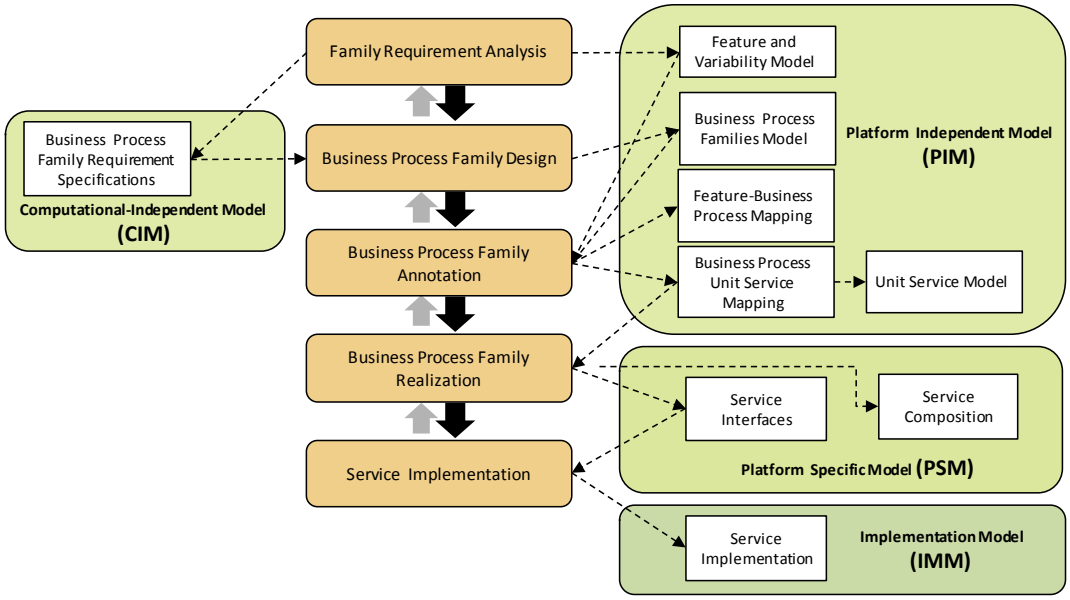


Figure 1. Domain engineering lifecycle for developing business family process

3. Method Overview

This section provides process-centered overview of our proposed method. We describe the main phases of the process with artifacts either used and or produced. This method has two main lifecycles: domain and application engineering. During the development process of these parts, we follow MDE principles.

3.1 Domain Engineering

Domain engineering aims at investigating a family of business processes and producing reusable assets and reference architectures of families of software. It also is responsible for scoping the product line and describing variability models of product lines. The main phases of domain engineering lifecycle are as follows (see Figure 1**Error! Reference source not found.**):

The *Domain Scoping* phase deals with the scoping of a business process product line and its market strategy [9]. That is, it identifies business functions which belong to a current product line (i.e. business process family). For this purpose, first a set of criteria such as number of potential business functions, range of variability, etc. are developed, and then used for scoping the business process product line [10]. This phase also produces business process product-line road-map as the output.

The *Family Requirement Analysis* phase aims at defining a model of requirements, where each requirement has a unique and unambiguous definition. Knowledge of existing business processes, customer viewpoints, project vision and documents, and the business cases serve as the inputs to this phase. Main activities performed in this phase are: *capturing user requirements* (which includes analyzing needs of users and stakeholders and then documenting them); *refining requirements* (which includes aggregating, decomposing, and alternating the requirements); *developing requirements model* (which uses requirements documents to define a model of the requirements, depicting the business functions and the non functional requirements of the business process family); *validation and verification* (that is making sure which requirements are valid and consistent); and finally *Developing the Feature Model* (which manages commonalities and variability within the product line and represent them in a feature model).

The *Business Process Family Design* phase takes the requirements model and the feature model and produces business process family model. This phase encompasses following activities: *business process activity identification* (which identifies, groups, and describes activities compelling satisfaction of some requirements [11]); *activities refinement*, (which decomposes activities into sub-activities); *dependency definition* (which describes conditions of each activity and relation between them, for instance, some activities might be performed, if some conditions are met); and *Developing Business Process Family Model* (which produces a business process model in one of the modeling languages such as BPMN).

The Business Process Family Annotation Phase aims at mapping the solution domain, the business process family model, to the feature model. It receives the feature model and the business process family model and produces a feature-business process mapping. Activities of this phase consist of: *tracing business sub-processes and requirements* (which identifies what parts of the business model realize the specific requirements); *binding business sub-processes* (which maps business sub-processes to the features encompassing corresponding requirements); and

Developing Unit Service Model (which defines unit services from business process models and specifies the type of activities as human-tasks and automated tasks [2][8][2]). The business process family annotation and business process family design phases can be performed in an iterative and incremental fashion. For instance, when a business sub-process related to requirements that belong to each feature is created, activities of this phase are performed and mappings between features and corresponding business process models are defined.

The *Business Process Family Realization* phase aims at implementing the business process model. It receives the business process family and unit service models and according to the type of activities, different implementations are developed. For each unit service, if it is a human-task, execution steps are defined, and if it is an automated task, the system tries to discover and compose service interfaces which implement that unit service with respect to requested non-functionalities.

The *Service Interface Implementation* phase is about the implementation of service interfaces, which provides the functionality of Unit Services accomplishing the activity defined in a business process. The service interface implementation is performed by means of service component development.

3.2 Application Engineering

Application engineering creates final products by adapting the reference architecture and by utilizing reused artifacts, created in domain engineering, with regards to the requirements of users of a specific product. It is a process of selecting a right set of business processes and their activities, software artifacts, and deployments in accordance with the requirements of the target business organization. During this lifecycle, requirements of specific applications are analyzed and the reference architecture is adapted and enriched with reusable artifacts to produce final products. The process of application engineering is illustrated in Figure 2. The main phases of application engineering are described below.

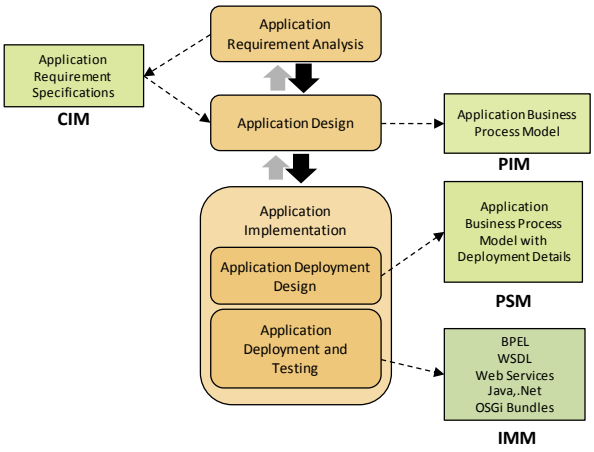


Figure 2. Application engineering lifecycle

The *Application Requirement Analysis* phase aims at collecting the requirements of the target business organization. It receives the project definition of the target business process and produces the requirements which will be used for choosing features from the business process family feature model. Activities of this phase are similar to activities of the family requirements analysis, but with the focus on a single application.

The *Application Design* phase receives application requirements and starts by selecting a configuration from the feature model and creating an initial business process model from the model describing the business process family. The initial business process model is selected according to the chosen features from the requirements analysis. At this point, we plan the use of different feature model specialization methods, which might be assumed as *Staged Specialization*. Our initial work on this matter is described in [17]. Furthermore, the selected feature configuration drives the selection of software artifacts, which implement the business process activities connected with the selected features. Software artifacts are software components implementing local services and external services. The business processes and the supporting software artifacts are represented through a BPML model. Software artifacts can be presented either with explicit naming, with the usage of text annotations, or as BPML artifacts. Herewith, the business process model can be verified from the perspective of stakeholders (e.g. business process engineers and designers), business process participants (e.g. knowledge workers, employees performing business activities) and software system engineers.

The *Application Implementation* phase is the final phase of service-oriented product development. The outcome of this phase is deployed and tested as a business process, and new business processes get implemented in the target application. This phase can be divided into two sub-phases. The first sub-phase is the *Application Deployment Design* sub-phase. During the deployment design, supporting software artifacts chosen in the previous phase and the deployment environment get integrated into the business process model. This model also has to be verified by the stakeholders, business process participants, and software system engineers, because it is the most accurate model of the business processes execution in the organization. The *Application Deployment and Testing* sub-phase, as the second sub-phase, begins after the approval of previously mentioned models. In this phase the business process is deployed and, the specified business process eventually is implemented in the organization. The three phases of application engineering are in their nature iterative and incremental. In the *Application Design* sub-phase, the features are being chosen according to requirements. In parallel to choosing features, business process models are automatically configured. During verification of configured business process models by stakeholders, business process participants, and software engineers; features can be selected and deselected. The selecting and deselecting of features may roll the process back to the *Application Requirement Analysis* phase, where these additional considerations affect the requirements of the business organization. Similarly, the iterative and incremental relations exist between the *Application Deployment Design* phase, and the *Application Design* phase. An example of changing the application design according to the results of the verification of business process models with integrated deployment information could be dissatisfaction of performance goals. Often there are cases where performance goals cannot be satisfied due to the deployment constraints.

4. Detailed Definition of Artifacts

This section describes detailed definition of main artifacts created within our methods. Artifacts, separated in different levels of abstraction, are created in domain engineering and application engineering lifecycles of the proposed method. In this section because of limited space we just introduce the domain engineering artifacts. These artifacts are categorized in three levels of PIM, PSM, and Implementation Models.

4.1 High-Level Design and Abstraction Model

The Platform Independent Model (PIM) contains models which are independent from a specific platform. A platform in a business process is an environment used to run business processes such as workflow management systems and SOA. This model encompasses features and variability models, families of business process models, unit service models, feature-business process mappings, and business process-unit service mappings. We provide brief descriptions for each of these models.

4.1.1 *Feature and Variability Model*

Feature modeling is the activity of identifying distinctive and prominent characteristics of software products in product lines. Features may be any distinguishable concepts or characteristics visible and exposed to various stakeholders (analysis, designer, developer, and users) [13]. Accordingly, in a business process family, a process can be represented as a visible characteristic which provides business functionality. Furthermore, a family of business process shares common features which provide and encapsulate business functionalities. Hence, modeling of variability of features in a business process family promisingly elevates software reuse through systematic exploitation of commonality and variability between processes.

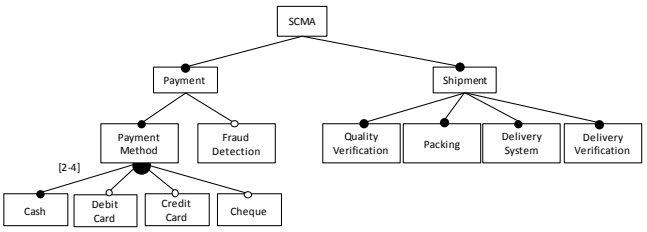


Figure 3. The feature diagram for configuration of a family of business process

Since, the notion of development software product families encompasses development of products sharing a significant amount of features based on common characteristics; variability modeling is used in all phases of the development process, particularly for business process management.

The effects of product family variability modeling are characterized and discussed at different levels in [16][14][15][16]. The development of a family of business processes embodies variability as a required concept which enables the derivation of distinct processes defined in a business process family. The ability to derive and configure business processes is achieved through using the variation points which generally define the decision points in sub-processes. Feature models are employed to represent the variability and describe the permissible configurations of a business process family. Common features among various sub-processes are modeled by mandatory features. Mandatory features are features which must be included in the final configuration and presented for the final product to function as intended. To exemplify this, let us consider a feature diagram for the Section 2.3, depicted in Figure 3. The feature model diagram contains features corresponding to sub-processes of the family of business processes. Each business process configured from this family must have the *Payment* and *Shipment* sub-processes. For example, *Payment* of the application with respect to annotation cardinality must include 2 to 4 methods of payment as mandatory features represented in feature diagram. Optional sub-processes may or may not be included for a family of business processes to function. For example, *Fraud Detection* is demonstrated as an optional sub-process in a business process family in order to support versatile security functionalities in different levels.

4.1.2 *Business Process Family Model*

After performing the requirement analysis for the business process family, a suitable business process should be implemented to fulfill these requirements. The business process family model is the bridge between requirements and implementation transforming the requirements to the implementation phase smoother. The business process family model is the result of the business process family design phase. A business process family model is the output of formalizing the informal description business process by using a concrete business process modeling language such as BPMN. This model encompasses all aspects of the target organization. Therefore, the modeling language should provide constructs and mechanisms to support aspects such as activity, sequence flow, dataflow, and events. Since, the information related to the target platform is not embedded in this business process family model, it can be considered as part of the PIM. Figure 4 depicts a business process consisting of two sub-processes denoted as *Payment and Shipment*. The business process starts when an *Order Request* message arrives at the business organization. After receiving the *Order Request* message, the payment is performed. The business process proceeds to *Shipment* when the request item is paid. After the completion of the *Shipment* the business process ends.

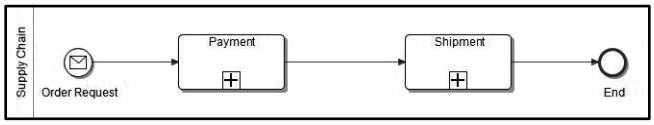


Figure 4. Process of supply chain family consisting of two sub-processes

Figure 5 shows the details of each sub-process from Figure 4. The *Payment* sub-process specifies how the payment for the required artifacts can be carried out. It can be carried out by using *Cash*, *Credit Card*, *Cheque*, or a *Debit Card*, represented as sub-processes in this model. Different alternatives of payment are mutually exclusive, that is, only one method of payment can be chosen at run time for this application, whilst the *Fraud Detection* process is under way. Fraud detection is responsible for checking and managing the fraud in payment.

The *Shipment* sub-process starts by acquiring goods in order to check whether goods are available and to verify the quality of goods. Then goods are packed and delivered to customers. If

there is problem in delivering goods, it will be managed by invoking the reclamation handling sub-process.

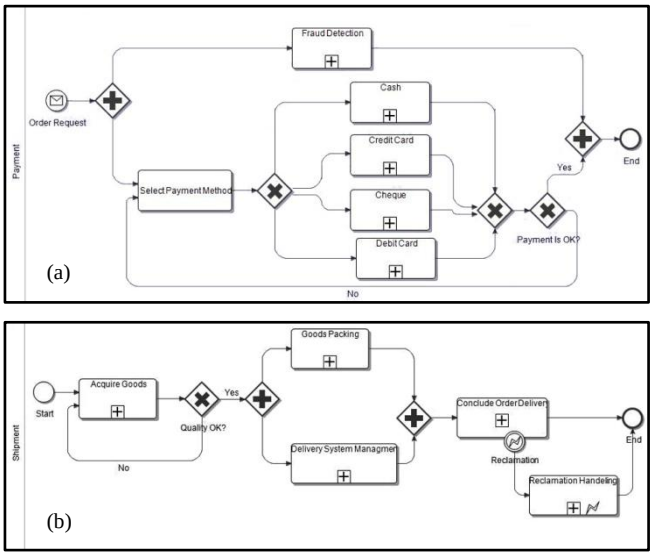


Figure 5. Detailed sub-processes of supply chain business process. (a) Payment sub-process. (b) Shipment sub-process

4.1.3 *Unit Service*

A family of business processes comprise of a coarse-grained sequence for flows of activities as conceptual units expressing the functions of business processes. In the SOA paradigm, in which a business process is composed of services, activities can be implemented by services or delegated to users. We consider a unit service as a formal definition of an activity in a business process. Thereby, by considering an individual activity as an atomic unit in a business process, an activity is performed by one or more unit services by following the principles of SOA. For each activity in a business process family, there might be one or more unit services which can perform and complete an activity in a business process. Moreover, unit services may be common and reused among business processes. Different unit services may provide business functionality with support for different technological requirements, such as communication over different transport mediums. Furthermore, they can provide business functionality with different non-functional properties such as security and performance, exposed by different implementations of service interfaces and composed services. The variation points, derived from non-functional and technological properties of service units, provide business functionality with different non-functional and technological properties. Such variability facilitates flexible quality-driven business process family configuration. For instance, the *Debit Card* sub-process includes an activity represented as *Receive PIN* in Figure 6. Different unit services might be discovered and mapped to an activity so as to provide secure transaction when a PIN is used (e.g. card reader, ATM keypad operating device, and etc.).

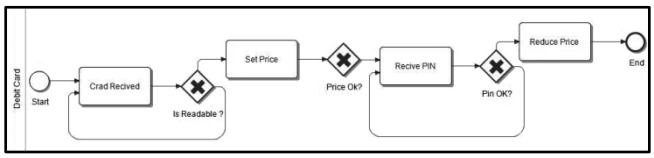


Figure 6. Activities defined in Debit Card sub-process

4.1.4 *Feature-Business Process Mapping*

The final software product is a software system having features selected in the final feature model configuration. The realizing software artefacts are selected from the solution model automatically with the feature selection.. The solution model consists of design and implementation models. In order to facilitate such production of the final software product,the relation between artefacts in the solution model (i.e. class diagrams, state diagrams, code, and etc.) and features in the feature model which they implement, must be defined. Different SPLE approaches adopt different policies for defining mappings between features and solution models. For instance, Gomaa [11] sets this mapping at the first stages of domain engineering (i.e. requirement engineering) and models features as groups of use-cases that are reused together. Consequently, since other artefacts, created in the following stages of development, are traceable to use-cases, this mapping propagates to other artefacts. On the other hand, another policy specification approach is used in FeatureMapper [18]. This approach provides separate stages and tools to map features in a feature model, as the problem space elements, to software artefacts, as solution model modules. This mapping should be maintained with structure to produce concrete products from configuration.

Since, business processes define our solution space, we need to have proper mapping between features and business processes, representing each feature realised by corresponding sub-business processes. We consider a mapping policy similar to the one used in the FeatureMapper [18] and also aim at developing a mapping tool to assist developers in creating this mapping for their family of products. Furthermore, feature-business artefact in our method is structured for maintaining mappings between features and sub-business processes. For instance, the *Debit Card* feature in feature model is mapped to the debit card sub-process of business process family (See Figure 6).

4.2 Platform Specific Model

The Platform Specific Model (PSM) comprises models which rely on specific platforms conforming to most common SOA platforms, enabling assembly and orchestration of loosely coupled services. During a domain engineering stage, Business Process Family Realization consists of implementing the business process model through unit services, as reusable assets in a domain, including different service interface implementation and composition.

In business process family development, either in domain or application engineering stages, the discovery and selection of available unit services is not automatic and requires intensive human efforts. Hence, to gain high level of reuse in a specific target platform, the identification and discovery of unit services approaching the distinct goals of activities in a business process is a significant task. As a possible solution, we aim at developing semantic service discovery to provide seamless and flexible discovery of unit services which provide the business

functionality required in a business process family. The discovery engine can retrieve a set of available unit services annotated by semantic descriptions in a specific platform, matching with activity goals in a business process. For this purpose, we would employ the Web Service Modeling Ontology (WSMO) [22] aiming at describing all pertained aspects related to general services and goals..

4.3 Implementation Model

Upon creation of the platform specific model, the constructional elements identified as requirements for the final software system need to be addressed by assigning proper technological pieces to each constructional element. In our implementation model, the required and existing technology for each of the artifacts from the PSM are identified and mapped to elements that guarantee a sound deployment of the technology elements on a low-level programming language platform. The implementation model enables the derived model to be easily compiled to the target programming language platform so that the final outcome complies with the requirements of the software system identified in the earlier steps of design and development.

There might be several implementation models that correspond to one service interface model. This stems from the fact that the implementation model represents a more concrete implementation of a service and thus requires to be closely bound to the underlying platform on which the final set of services will be deployed. The variety of deployment platforms requires more concrete concepts to be considered at the level of service implementation compared to service interface, which in turn may lead to various instances of implementation model for the service interface model. Finally, the service implementation model gets compiled to a set of services and components in a target programming language, which once compiled, can generate the final software system. It is important to note that the problem of composing services and components into a software system is addressed at the upper layers of PIM and PSM software design.

Provoking SOA as the main principal in our model driven software engineering design, we map the final elements and artifacts of our implementation model to software entities representing self-contained services or components that deliver the desired functionality corresponding to the required artifacts in the implementation model. These software artifacts range from service descriptions (e.g., WSDLs) and their corresponding concrete implementation for Web services to service component modules (e.g., SCM components) to lower level self-contained components encompassing the desired functionalities (e.g., OSGi bundles). The selection of the target implementation platform is very much influenced by the directives and decisions enforced at the higher levels of modeling and designing of the software system, i.e., when deciding about the business process representation of the system under design, or identifying its commonalities and variabilities.

5. Related Work

The PESOA methodology for the development of process families [19] is the most related work to our work. The PESOA approach aims at ameliorating the development of variant-rich processes models by representing variability in a BPMN based process family architecture. In the analysis phase of the established methodology, the requirements of an e-business system to be developed as well as the stakeholder's needs are captured, and variabilities are presented by feature models. A business process model is derived manually from a feature model which represents the variability in business processes. The authors' approach for capturing variability in process models relies on extending UML Activity Diagrams and BMPN models by stereotype annotation. Accordingly, the variability is accommodated in the process model in a so-called variant-rich process model. The variability points captured are associated to features in order to achieve feature configuration. In other words, features in the generated feature model are actually direct representations of the variants among processes. The implementation phase in the proposed methodology focuses on development of product family implementation artifacts and deploying business process specifications into the process execution engine The PESOA approach captures only design-time variability and there is no possibility to modify the variants during run-time.

Montero et al. [20] also introduce an approach for the development of families of business processes. The intention of their approach is to make consistency between the business process model and the corresponding feature model and to introduce runtime variability. With respect to this idea they have recognized the feature model such that parent features, or variability points are considered as complex processes, and child features, are their sub-processes. They produce the basic business process from the feature model and complete the created business process manually. Since there are many relations in a business process, such as sequence, deferred choices and etc.; feature models cannot be employed for this purpose without appropriate declaration of semantic relations. Nonetheless, our work differs from theirs. We demonstrate the relations and integrity constraints among sub-processes in a family of business processes through feature modeling, and the implementation and behavior of sub-process are performed by business process models. Furthermore, in our approach, the proper requisite mapping between feature models and business process models is also defined.

Bae et al. [21] introduce the FMBP, a methodology for developing a feature model out of families of business processes. The FMBP approach adopts a top-down divide-and-conquer method for the development of feature models. This methodology argues for the development of a feature model from a business process model by identifying use cases from business process models and aligning features in accordance with them. This method is resembled to the method proposed by Montero et al. [20], yet, Bae et al. focus only on business process modeling. They also have the problem of difference of semantic relation in feature model and business process due to converting business process to feature model.

6. Conclusion and Future Work

In this paper we presented a novel methodology for the development of business process families by exploiting SPLE and SOA. In our methodology we introduce variability modeling derived from different levels of SOA development to support a high level of reuse and to facilitate the development of variant-rich business process models. We have described how a family of business processes can be modeled and variability can be captured in different development stages to approach the flexible and cost-effective development and deployment of a family of software products. Furthermore, with the comparison of current approaches for model-driven development of semantically rich

business processes and supporting SOAs, we described how we improve the state of the art in model-driven development of families of SOAs. We have also made the initial steps towards realization of supporting tools for our vision.

In our future work, we will empower different stages of development phases by taking advantages of emerging semantic web technologies. We employ ontologies to define semantic relations between different software artifacts and models. We will exploit ontologies as an underlying formalism for the representation of feature models of families of business processes to incorporate semantic annotation of features in order to approach semi-automatic configuration of a family of software services. As part of our future work, we will devise to develop semantic service discovery supporting tool to identify and discover unit services in which we perform activities based on defined business goals and requirements imposed by activities in a business process model.

7. Reference

[1]Tsai.W., 2005. Service-oriented system engineering: a new paradigm, *IEEE International Workshop on Service-Oriented System Engineering, SOSE 2005*. pp. 3-6.

[2]Chang, S. H, and Kim, S. D., 2007. A Variability Modeling Method for Adaptable Services in Service-Oriented Computing, *In Proceedings of the 11th International Software Product Line Conference, SPLC 2007.*, pp.261-268.

[3]Clements, P., Northrop, L., 2001. Software product lines, Addison-Wesley Reading MA.

[4]Bezivin, J., Dupe´, G., Jouault, F., Pitette, G., and Rougui, J. E.. First Experiments with the ATL Model Transformati on Language: Transforming XSLT into XQuery, *In Proc of the Workshop on Generative Techniques in the Context of Model Driven Architecture*, Anaheim, CA.

[5]Weske, M.. Business Process Management, Springer, 2007.

[6] Mendling, J.. 2008. Business process management, *Lecture Notes in Business Information Processing*, Springer.

[7]List, B., and Korherr, B., 2006. An evaluation of conceptual business process modelling languages. *In Proceedings of the 2006 ACM Symposium on Applied Computing* (Dijon, France, April 23 - 27, 2006). SAC '06. ACM, New York, NY, 1532-1539.

[8]Sasa, A., Matjaz, B. J., Krisper, M., 2008. Service-oriented framework for human task support and automation. *IEEE Transactions on Industrial Informatics.* Nov. 2008, vol. 4, no. 4, p. 292–302.

[9]Linden, F. J., Schmid, K., and Rommes, E., 2007. Software Product Lines in Action , Springer,

[10]Kim, S., Min, H. G., Her, J. S., Chang, S. H., 2005. DREAM: A practical product line engineering using model driven architecture. *In Proceedings of the International Conference on Information Technology and Application. 2005b*, Australia, pp.70-75.

[11]Gomaa, H., 2004. Designing Software Product Lines with Uml: from Use Cases to Pattern-Based Software Architectures. Addison Wesley Longman Publishing Co., Inc

[12]Papazoglou, M. P., and Yang, J., 2002. Design Methodology for Web Services and Business Processes. *In Proceedings of the Third international Workshop on Technologies For E-Services (August 23 - 24, 2002).* A. P. Buchmann, F. Casati, L. Fiege, M. Hsu, and M. Shan, Eds. *Lecture Notes In Computer Science, vol. 2444*. Springer-Verlag, *Londo*n, pp.54-64.

[13]Jaejoon Lee, D. Muthig, and M. Naab, 2008. An Approach for Developing Service Oriented Product Lines, *In 12th International Software Product Line Conference, SPLC '08.* pp. 275-284.

[14]Halmans, G., Pohl, K., 2003. Communicating the variability of a software-product family to customers, Software and Systems Modeling, vol. 2, pp. 15-36.

[15]van Gurp, J., Bosch, J., and Svahnberg, M, 2001. On the notion of variability in software product lines". *In proceedings of the IEEE/IFIP Conference on Software Architecture*, pp. 45-54.

[16]Bachmann, F., Bass, L., 2001. Managing Variability in Software Architecture. ACM Press, NY, USA, 2001

[17]Mohabbati, B., Kaviani, N., Gašević, D., 2009. Semantic Variability Modeling for Multi-staged Service Composition, *In Proceedings of the 13th Software Product Lines Conference, Vol. 2 (3rd International Workshop on Service-Oriented Architectures and Software Product Lines)*, 2009 (in press).

[18]Heidenreich, F., Kopcsek, J., and Wende, C., 2008.FeatureMapper: mapping features to models, *In Companion of the 30th international Conference on Software Engineering (Leipzig, Germany, May 10 - 18, 2008)*. ICSE Companion '08. ACM, New York, NY, 943-944.

[19]Schnieders A, and Puhlmann F, 2006. Variability mechanisms in e-business process families, *9th International Conference on Business Information Systems* (BIS 2006), 2006.

[20]Montero, I., Pena, J., Ruiz-Cortes, A,, 2008. From Feature Models to Business Processes, *In Proceedings of the IEEE International Conference on Services Computing Vol. 2*, pp.605-608.

[21]Bae, J. and Kang, S. A, 2007. Method to Generate a Feature Model from a Business Process Model for Business Applications, *In Proceedings of the 7th IEEE international Conference on Computer and information Technology (October 16 - 19, 2007)*. CIT. IEEE Computer Society, Washington, DC,. 2007, pp. 879-884.

[22] Roman, D., Keller, U., Lausen, H., de Bruijn, J., Lara, R., Stollberg, M., Polleres, A., FeierC,Bussler, C., Fensel,D. ,2005. Web service modeling ontology. Appl Ontol 1(1):77–106.

[23]OMG, Business Process Modeling Notation specification2.0,http://www.omg.org/technology/documents/bms_spec_catalog.htm.

[24]van der Aalst, W., Ter Hofstede, A., Weske, M., 2003. Business Process Management: A Survey. International Conference on Business Process Management (BPM 2003), *Lecture Notes in Computer Science volume 2678*, pages 1-12. Springer-Verlag, Berlin.

[25]Booch, G., Brown, A., Iyengar, S., and Selic, B.: An MDA Manifesto. MDA Journal (2004)

Interaction-based Feature-Driven Model-Transformations for Generating E-Forms

Bedir Tekinerdoğan
Bilkent University
Dept. of Computer Engineering
06800 Bilkent, Ankara, Turkey

bedir@cs.bilkent.edu.tr

Namik Aktekin
eMaxx B.V. Hengelo,
P.O. Box 768,
7550 AT, Hengelo, The Netherlands

n.aktekin@exxellence.nl

ABSTRACT
One of the basic pillars in Model-Driven Software Development (MDSD) is defined by model transformations and likewise several useful approaches have been proposed in this context. In parallel, domain modeling plays an essential role in MDSD to support the definition of concepts in the domain, and support the model transformation process. In this paper, we will discuss the results of an e-government project for the generation of e-forms from feature models. Very often existing model transformation practices seem to largely adopt a closed world assumption whereby the transformation definitions of models are defined beforehand and interaction with the user at run-time is largely omitted. Our study shows the need for a more interactive approach in model transformations in which e-forms are generated after interaction with the end-user. To show the case we illustrate three different approaches for generation in increasing complexity: (1) offline model transformation without interaction (2) model transformation with initial interaction (3) model-transformation with run-time interaction.

Categories and Subject Descriptors
D.2.2 [**Software Engineering**]: Design Tools and Techniques.

General Terms
Design, Documentation, Performance, Verification

Keywords
Model-driven software development, Feature-oriented modeling, e-government

1. INTRODUCTION
One of the basic pillars in Model-Driven Development is defined by model transformations and likewise several useful approaches have been proposed in this context [6][1]. In addition it should be noted that the goals of model-driven development also depend on the identification and modeling of the right domain concepts. As such domain analysis plays an essential role in MDSD to support the definition of concepts in the domain. Domain analysis is a systematic approach for analyzing and modeling the domain concepts that are relevant for the stakeholders [2]. One of the common techniques for domain modeling is feature modeling, which has been extensively used in domain engineering [2]. Hereby, a feature model is a result of a domain analysis process in which the common and variant properties of a domain are elicited and modeled. In addition, the feature model identifies the constraints on the legal combinations of features. A feature model can thus be considered as a specification of the family.

In this paper, we report on our experiences of applying feature modeling to model-driven development. The context of the case is an e-government project which aims to use information and communication technology to provide and improve government services. E-government includes different models including government-to-government and government-to-citizen. We have focused on the model of local government-to-citizen which aims to support the interaction between local and central government and private individuals. Part of the e-government solutions are the generation of e-forms (electronic forms) for local governments. An e-form is the electronic version of its corresponding paper form. We have applied model-driven engineering techniques for the automatic generation of e-forms (electronic forms) from feature models.

This project has shown that feature modeling is an effective means not only to model the domain of e-forms but also to support the automatic generation in a model-driven engineering process. Besides of this observation the results of our study also presents an additional insight and lessons learned regarding model transformation practices in general. In particular it appeared that for defining e-forms offline static single generation is less suitable. This is because the specific e-forms depend on the user input and the retrieved data from the data administration services. In this paper we show three different approaches for generation with increasing complexity: (1) off line model transformation

without interaction (2) model transformation with initial interaction (3) model transformation with run-time interaction. We report on our experiences and lessons learned and propose a systematic approach for defining model transformations that is based on an interactive paradigm.

The outline of the paper is structured as follows: In section 2 we provide the case study on e-form generators for local governments. In section 3 we show the automatic transformation process for generating e-forms from feature models. In section 4 we present an interaction-based model transformation. Finally section 5 presents the conclusions.

2. CASE STUDY – E-FORM GENERATION

2.1 Description

The research has been carried out together with eMAXX which is a medium-size ICT company in Enschede, The Netherlands [5]. One of the objectives of eMAXX is to produce solutions for e-government (electronic government). Figure 1 shows an example interface of e-government gateway of the city Enschede, which the citizens can access to request services.

Figure 1. Example interface of a local government interface for supporting e-services

An e-form is simply the electronic version of its corresponding paper form. E-forms have some benefits over paper forms including eliminating the cost of printing, storing, and distributing pre-printed forms. In addition e-forms can be filled out faster because the programming associated with them can automatically format, calculate, look up, and validate information for the user. With digital signatures and routing via e-mail, approval cycle times can be significantly reduced. Compared to paper forms, e-forms allow more focus on the business process or underlying problem for which they are designed (for example, expense reporting, purchasing, or time reporting). They can understand the roles and responsibilities of the different participants of the process and, in turn, automate routing and much of the decision making necessary to process the form.

Using e-forms on the internet site of the local governments, citizens can perform requests such as making an appointment, informing about a movement, requesting a build license, etc. These services are defined on e-forms that are implemented by eMAXX. The deployment view is depicted in Figure 2.

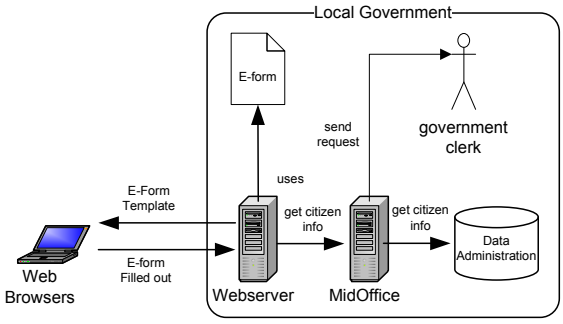


Figure 2. E-form generation –manual case

E-forms remain at a web server of a local government. Citizens can access these web pages through the internet browsers. E-forms are usually defined over multiple web pages. Once a user logs in to the system the user can select a number of services offered by the local government, such as for example *notification of movement*. A middleware layer, defined by the *MidOffice* server includes functions to access personal data of the registered users in the local government, which is stored in one or more back office systems, *Data Administration*. Based on the selected product and the user, the information about the user is requested from the common administration through *MidOffice*. The unknown fields are filled out by the user. After the citizen enters the last field the system needs to generate a report and submit the request to the *government clerk*. An important advantage of the *MidOffice* is the loose coupling between the interfaces (presentation) and the back offices (data). Different back office system can be accessed by different web browsers. The communication of the client web pages only communicate through the *MidOffice* which is responsible for the communication and distribution logic.

2.2 Problem Statement

In the initial version of the system, e-forms were manually implemented and deployed on the webserver of local governments. Moreover, e-forms are statically defined without taking into account the interaction with the user. A number of problems with this manual, static development solution can be identified.

- *Lack of reuse of e-forms*

First of all, even when we are dealing with the same kind of service, such as *notification of movement*, different local governments might require different kind of e-forms. The differences might be in the required type of data, the presentation form or the control flow i.e. the order in which the data is presented to the citizen. Although the e-forms share much commonality, the lack of systematic variability management requires that for each different local government an e-form needs to be implemented from scratch.

- *Maintenance of e-forms*

Even after deployment of the e-forms on the web servers, based on earlier practical experiences, updates might required to the implemented e-forms in due time. Unfortunately, the maintenance of the web pages including the e-forms is not trivial and again requires changes to the requested data, the presentation form or the control flow.

- *Need for run-time generation of e-forms*

Since the generation of some fields can only be known when a particular citizen is filling out the e-form, the specific required e-form can actually only be known at run-time. Because of this limitation usually the complete e-form is provided to the user, which complicates the process of filling out the form by the citizen. The e-form would be easier if only the required information is presented at the right time.

- *Need for interaction by user*

Finally, related to the previous third issue, when filling out the e-form, interaction with the *Data Administration* might be required to retrieve data to speed up the process or to complete the e-form. Unfortunately, in the initial version the interaction is only defined in the beginning during the authentication step of the citizen.

Regarding the above issues the manual implementation of e-forms with only weak interaction with citizen and/or back offices is to some extent doable but certainly not cost effective. To optimize the development, maintenance and usage of e-forms automated support is necessary. The main objective here is to increase the reuse and productivity while developing and maintaining e-forms. For this, two basic issues need to be addressed. First of all, a domain model is required for defining e-forms. This domain model should be easy to understand and to be developed. Secondly, based on the domain model the target artifacts, that is, e-forms need to be automatically generated. To address these issues we have defined three different types of generators in increasing complexity:

- *Generator without interaction*. This generator transforms a feature model to an e-form in which all the required fields are presented to the end-user. The end-user needs to fill out all the requested data and the e-form can only be completed if all the information is entered. Once the e-form is complete a report is generated and the service request is submitted for handling.
- *Generator with single, initial interaction*. This generator is similar to the previous generator but allows for initial interaction with the data administration server to retrieve the values for the fields that can already be defined in the e-form
- *Generator with multiple, run-time interaction.* This generator complements the second generator by allowing interaction with user and data administration during run-time. For this a number of functions of data administration can be invoked to speed up the e-form completion process. Because of the multiple options for invoking functions the generator defines the related workflow for optimizing the function calls.

Obviously, explicitly addressing interaction in model transformations is here a key issue. Unfortunately, current model-driven development practices tend to adopt a more closed-view approach in which interaction is not explicitly addressed. Our experiences in this industrial context aim to show both the necessity for interaction in model-transformations and the role of feature modeling. In the following sections we elaborate on the above generators.

3. FEATURE-BASED MODEL TRANSFORMATION

To address the requirements in the previous section we (1) define feature models of local governments, and (2) use these to generate e-forms and reports. Feature models have thus a dual role of modeling the data and as an intermediate form of e-forms. In the following we will discuss the first generator process which automates the e-form generation process but does not include interaction. In section 3.1 we will first focus on feature modeling of the services, and in section 3.2 we will discuss how we adopt and integrate feature models in the model-transformation process.

3.1 Feature Modeling of Services

Different e-forms are implemented for different local governments but besides of the variations one can easily observe commonality of requested data. To model the domain for a given service we define a *family feature model*. Figure 3 shows, for example, a feature diagram for a service of a local government, which is the *notification of moving*. In fact this feature model defines the space of requested data that can be implemented on different e-forms. To put it differently, the feature diagram represents an intermediate representation for the space of e-forms.

The feature model is already useful for supporting the implementation of e-forms. Different instantiations of the feature diagram indicate different definitions of e-forms. An example instantiation of the family feature diagram in Figure 3 is given in Figure 4.

Based on the application feature model the corresponding e-form can be implemented. Herewith, all the mandatory features will need to be mapped to fields. Optional and alternative features will be for example realized using check box fields, or radio buttons. We have defined a set of transformation rules and implemented these in the transformation definition. A possible corresponding e-form is depicted in Figure 5.

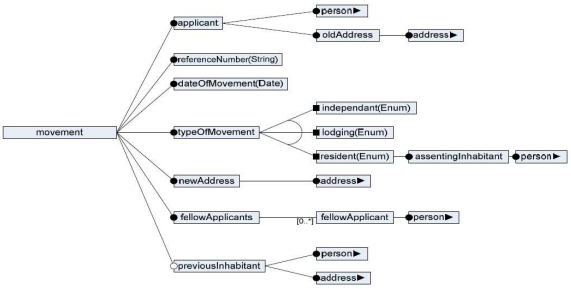


Figure 3. Family feature diagram of service *notification of movement*

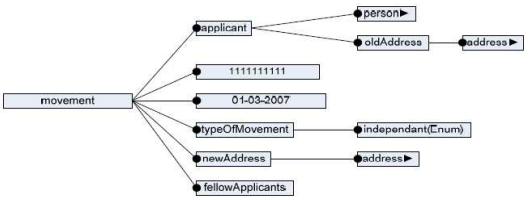


Figure 4. Application feature model of service *notification of movement*

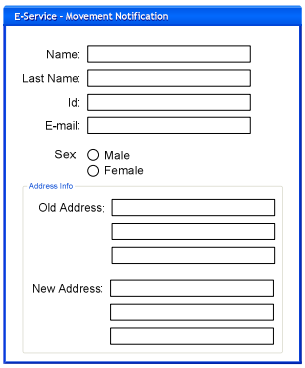


Figure 5. Example E-form based on instantiated feature diagram

3.2 Model Transformations

In principle, feature models can be used for manually implementing e-forms. However, to support reuse and productivity, we will aim for automatic generation of e-forms. For this a generator needs to be defined that takes as input an instantiation of the feature model and provides as output the corresponding e-form. As defined in Figure 2, after the citizen fills out the e-form a report needs to be generated and the request should be handled. Obviously, here we can easily apply model-driven techniques to support the reuse and automation goals.

For the given case, at least three different transformations are required as defined in Figure 6:

⓪ Defining feature model – The domain modeler defines a feature model of the required services. This is a manual process.

① *Application feature model* to *UI model* – The instantiated feature model of the service, the application feature model, will be used to generate the UI model representing the e-form.

② *UI model* to *feature model* – once the user fills out the required fields in the form the UI model will be generated to the feature model.

③ *Feature model* to *Report model* – After all the fields in the e-form are filled out, and the final feature model is generated, a report will be generated.

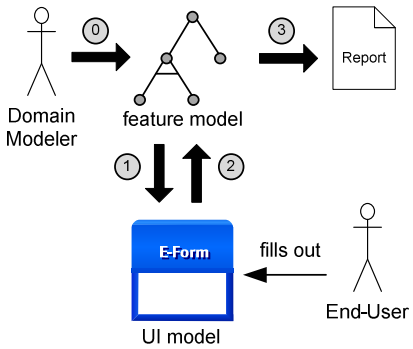


Figure 6. Required transformations for automatic e-form generations

Figure 7 shows the transformation pattern for generating e-forms based on feature models. For defining the model transformation we need to define the source metamodel, the target metamodel and the transformation definition. In fact, both the source model and target models are known. The source model, FM1 in Figure 7, is a feature model, that conforms to a feature metamodel MMFM, which defines the common concepts for feature models. We have adopted the metamodel as defined in [3]. The target model UI defines e-forms, and conforms to a metamodel MMUI. All the models are represented using XML. The transformation applies XSLT which is a language for transforming XML documents to other XML documents. All the models in Figure 7 conform to the metametamodel MOF.

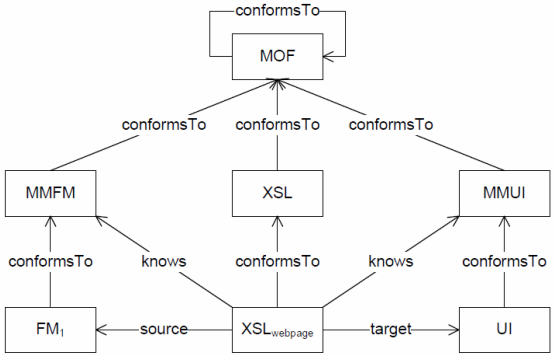


Figure 7. Transformation pattern for transforming feature model to UI model (e-form)

Once the citizen has filled out all the fields the final instance of the feature diagram will be defined, requiring a transformation from UI model to feature model. This is in principle similar to the transformation pattern as shown in Figure 7, only the source model will now be the UI model and the model the feature model.

Figure 8 shows the transformation pattern for generating reports based on e-forms. Since the e-form is represented as a feature model the source metamodel is a feature metamodel MMFM, and the target metamodel is a metamodel for describing reports, MMR.

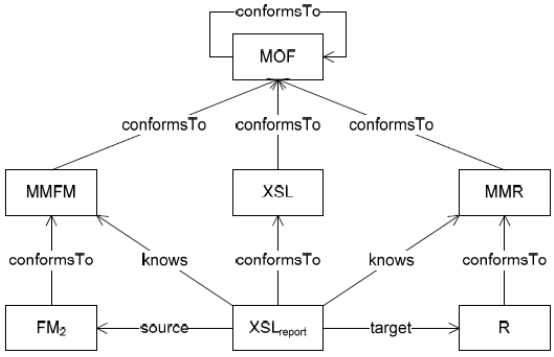


Figure 8. Transformation pattern for transforming feature model to Report

To sum up, this generation process automates the e-form development process by using feature models. The complete e-

form is generated and presented to the citizen. Once the citizen has completed the e-form the report can be generated.

4. MODEL TRANSFORMATION WITH INTERACTION

In fact the overall model-driven process in section 3 largely supports the goals for automated development of e-forms. However, the transformation process in section 3 does not take into account interaction with the user and the data administration. The generated e-form is actually statically defined in one step, one web page is generated, and no interaction is possible with the end-user or data administration. In fact all the transformation steps in Figure 6 are executed once. In the following sections we will define generators that include interactions with the user and data administration.

4.1 Initial Interaction

The second more refined generator makes use of the calls to the data administration. After the authentication process and selection of a particular service the system can already retrieve some information about the citizen and the selected product and instantiate part of the feature diagram. As such, the time to fill out the form, as well as the chance for incomplete forms will be partially reduced.

Compared to the generation process of the previous section this generation process includes one more transformation pattern. This is the transformation from a source feature model to another target feature model. As such the process of e-form generation requires the following order of steps:

1. Authentication of user
2. Selecting product service
3. Loading family feature model
4. Call to data administration to retrieve personal details
5. Definition of application feature model based on retrieved data in step 4
6. Generation of e-form based on application feature model
7. Entering data by user in the e-form of step 6
8. Transformation of e-form to feature model

4.2 Run-time Interaction

The first generator without interaction solves the automation problem of e-forms. By defining transformations e-forms can be automatically generated. The second generator allowed initial interaction with data administration to retrieve data that could be filled out. As such the e-form completion process time is reduced. However, both generators generate one complete web page in which all the fields are shown. Unfortunately, this is not always suitable since the generation of the specific fields in the e-form also depends on the data that is entered by the user, or the data is retrieved from the data administration, at run-time. As such, the third generator allows run-time interaction with the user and data administration. In this way, the e-form is generated incrementally dependent on the input of the end-user. This means that the instantiation of the family feature diagram is not done after authentication process but at any time during completing the e-form. Also multiple web pages including part of the e-form are generated.

The interaction process is shown in Figure 9. After the authentication process, the family feature model is retrieved and the first fields are defined. Then follows a cycle of interaction with user and data administration in which the application feature model and likewise the corresponding e-form is specialized. Once the e-form is complete a report is generated and the request is submitted. In essence the transformation process is similar to the alternative without interaction. The main difference is that now the feature model is specialized multiple times and during the e-form completion process. Obviously, multiple model transformations are required to complete the process. In fact, this process also follows the idea of staged configuration of feature models as explained in [3].

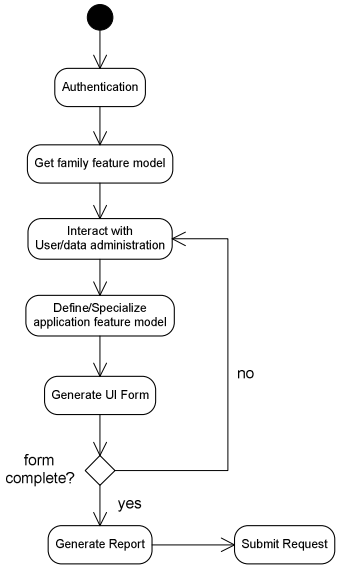


Figure 9. Transformation pattern for transforming feature model to UI model (e-form)

4.3 Optimizing Workflow

When interaction with the data administration is supported functions for data administration are accessed. Many different functions might be accessed given an application feature model. For example, the invocation of the function *getPersonDetails* can define the values for name, address, and id of the citizen. Further, each invocation of a function might result in the definition of the values of different fields.

In essence the aim is to optimize the e-form completion process and therefore the functions need to be preferably invoked in the order in which the maximum set of values in the e-form can be determined. The latter means that the number of fields that the citizen needs to enter is optimally reduced.

It appears thus that we need to address the workflow explicitly to optimize the generation process. In the first generator no data administration function was called at all. In the second generator only initial call was made to the data administration. As such the workflow concern was not considered in these two generators. In the third generator the workflow concern is explicitly considered by (1) defining the functions that can be invoked (2) defining the order in which they need to be processed. As such based on the state of the e-form (and the application feature model) a decision needs to be made which functions of the data administration need

to be called. Different strategies can be adopted for this. We have adopted a simple fixed, strategy which aims to optimize the number of model transformations needed. The workflow definition is defined as depicted in Figure 10.

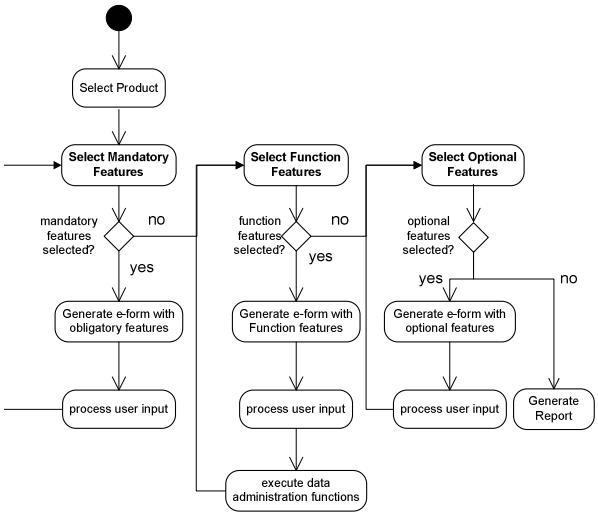


Figure 10. Adopted workflow in the interaction-based e-form generator

Hereby we first check whether mandatory features have been defined in the feature model. These are then first processed, that is an e-form is generated with these fields, and data input from the user is processed resulting in a new feature diagram. The following step is to select features that are related to functions in the data administration. The final step is the generation of optional features. Once all the fields have been entered the report is generated. In fact this is quite a simple workflow strategy and can be optimized in different ways. For example, we could prioritize the functions that result in more input from data administration; we could define the optimal path of these functions, etc. The full integration of strategy selection and optimization has been reserved for the future work.

5. CONCLUSIONS

In this paper we have discussed our experiences with using feature models for generating e-forms using model driven engineering techniques. The basic conclusion of this work is that an appropriate domain model represented as feature diagrams provides a solid basis for the space of alternative target models. In our case the target models were basically e-forms. Using the conventional model transformation pattern we have defined four different kinds of model transformations: feature model to feature model, feature model to e-form, e-form to feature model, feature model to report. All these transformations supported the automation process of e-forms and as such improved reuse and productivity. In addition we have pinpointed the necessity for interaction in generating e-forms. This is because the e-form is not only defined by the selected service but also defined by the entered answers in the e-form or the retrieved information from the data administration. To cope with this issue, model transformations could not remain static and/or offline but had to be integrated in the run-time e-form completion process. Based on the input at important steps in the e-form completion process the application feature model was regenerated and in accordance with this the e-form updated. It also appeared that hereby the order in which the functions of the data administration are accessed, i.e. the workflow, have an impact on the e-form completion process. In alignment with this issue, we have shortly discussed the notion of workflow concern. Our future work will focus on the interaction aspects in model transformations in general. We think that the lessons that we have derived from the considered project should be considered from a general and broader perspective. In particular the issue of interaction in the model-transformation process is a topic that needs further investigation.

ACKNOWLEDGMENTS

We would like to thank Mehmet Aksit, Anton Boerma and Richard Scholten for earlier support and discussions about this work.

REFERENCES

[1] J. Bézivin, F. Büttner, M. Gogolla, F. Jouault, I. Kurtev, A. Lindow. Model Transformations? Transformation Models!, MoDELS2006, Springer LNCS, Vol. 4199, pp. 440-452, 2006.

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

[3] K. Czarnecki, S. Helsen and U. Eisenecker, *Staged Configuration Through Specialization and Multi-Level Configuration of Feature Models*, Software Process Improvement and Practice, special issue on Software Variability: Process and Management", vol. 10, pp. 143-169, 2005.

[4] Eclipse Modeling Framework Web Site, http://www.eclipse.org/emf/

[5] eMaxx B.V. Hengelo, P.O. Box 768, 7550 AT, Hengelo, The Netherlands. http://exxellence.nl/

[6] D.S. Frankel. Model-Driven Architecture, Wiley Publishing Inc., 2003.

Towards Feature-driven Planning of Product-Line Evolution

Goetz Botterweck
Lero, University of Limerick
Limerick, Ireland
goetz.botterweck@lero.ie

Andreas Pleuss
Lero, University of Limerick
Limerick, Ireland
andreas.pleuss@lero.ie

Andreas Polzer
Embedded Software Laboratory
RWTH Aachen University
Aachen, Germany
polzer@embedded.rwth-aachen.de

Stefan Kowalewski
Embedded Software Laboratory
RWTH Aachen University
Aachen, Germany
kowalewski@embedded.rwth-aachen.de

ABSTRACT

Industries that successfully apply product line approaches often operate in markets that are well established and have a strategic perspective. Consequently, such organizations have a tendency towards long-term planning of products and product lines. Although there are numerous approaches for efficient product line engineering, there is surprisingly little support for a long-term, strategic perspective and an evolution of product lines. To address these challenges, we aim to integrate evolution into model-driven product line engineering. In particular, we explore how feature models can be applied to describe the evolution of product lines. The paper contributes (i) concepts for describing the evolution of product lines with feature models, (ii) a corresponding framework, which puts this into a bigger context and (iii) three scenarios that show how this framework can be applied. The concepts are motivated with examples from automotive software engineering and embedded systems, which are industries with a strong affinity to product lines, where long term planning of the product portfolio are common strategies.

Categories and Subject Descriptors

D.2.13 [**Software Engineering**]: Reusable Software; D.2.2 [**Software Engineering**]: Design Tools and Techniques

General Terms

Design, Algorithms, Management

Keywords

Product line engineering, Evolution, Feature modeling

FOSD'09, October 6, 2009, Denver, Colorado, USA.

1. INTRODUCTION

Since the first concepts on program families [17], software product lines [23, 4], and generative programming [5], approaches for the systematic engineering of families of software systems have come a long way and are successfully applied in many industries [20]. In recent years, we have seen an increase in *efficiency* in product line engineering (PLE), e.g., by using model-driven techniques [22] and improved techniques for the implementation and composition of features [3, 2, 12].

Industries that successfully apply product lines approaches often operate in markets that are well established and have a strategic perspective. Consequently, such organizations have a tendency towards long-term planning and evolution of their product portfolios. A typical example is the usage of embedded systems in the automotive industry, where a Systems Engineering approach with an integrated design of hardware and software (*"co-design"*) is applied and requires careful synchronization of the involved processes. This strategic perspective is reflected by a long-term planning of product portfolios on feature-level over several years. For instance, it is common to have decisions by upper management like "in 3 years we will introduce the next generation of our hybrid compact car" and "in 7 years we will offer hybrid drive for all cars in our portfolio".

Some PLE approaches acknowledge the importance of such an long-term perspective on the *motivational level* (e.g., by discussing the activity of scoping or by motivating PLE with strategic reuse [4]) but leave out tool-support. Other PLE approaches deal with evolution on the *detailed technical level*, e.g., to evolve feature implementations (see related work in Section 6), but leave out the long-term perspective and proactive planning of this evolution.

There is little support for *evolution on a model level*, which (1) provides an appropriate abstraction level to afford proactive planning and handling of complexity and (2) is precise and expressive enough to serve as a foundation for interactive and automated tools. In particular, to the best of our knowledge, there is no model-based support for evolution planning that allows capture evolution requirements ("In 5 years introduce feature x") in terms of models. Besides other drawbacks this prevents to feed such information

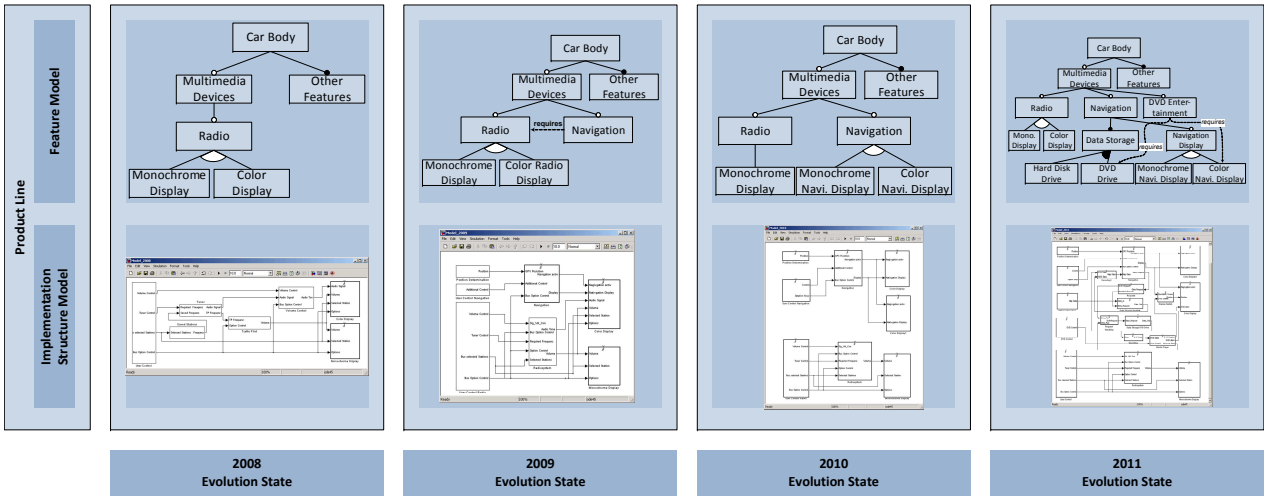


Figure 1: Example product line evolving over time.

into a model-driven workflow for PLE, such that future generations of the product line can be planned and derived.

Consequently, these existing approaches lack efficiency (because they do not support interactive and automated techniques) or fall short when it comes to handling the evolution of complex product lines (because they do not support the abstraction from details).

We intend to use feature models to handle the evolution of product lines. In this paper, we take first steps towards this goal by analyzing the research problem and defining our approach on a conceptual level: We motivate our research by examples from automotive embedded systems (Section 2). The paper contributes (i) concepts for describing the evolution of product lines with feature models (Section 3), (ii) a corresponding framework which puts this into a bigger context (Section 4) and (iii) three scenarios that apply this framework (Section 5). We conclude with a brief overview of related work (Section 6).

2. RESEARCH PROBLEM

2.1 An Example of an Evolving SPL

Figure 1 shows an example product line of automotive embedded systems, which provide entertainment and navigation features. This product line is evolving over four years from 2008 to 2011 (horizontal dimension).

In each evolution step the product line consists of various artefacts (vertical dimension). The *Feature Model* (upper layer) describes available configuration choices for a multimedia system for the core body of a car. In 2008 there is just an optional radio which can be configured. In 2009 a navigation system is added, which uses the user interface of the radio. Due to a management decision the navigation is separated from the radio system in 2010. As a consequence, navigation can be ordered separately and offered with the feature *Color Display*. In 2011 the navigation system is updated with the possibility to store more maps and *DVD Entertainment* is introduced, which requires a *DVD Drive*.

The bottom layer represents the implementation in an domain-specific language, here a Matlab/Simulink model. When products are derived from such a product line, negative variability [22] (i.e., the selective removal of elements in these models) can be used to automatically derive the implementation from a product-specific feature configuration.

2.2 Requirements for Modeling SPL Evolution

In the introduction and by providing the example we have motivated the need for a systematic handling of product line evolution. In the approach presented in this paper we strive to address this challenge by *modeling* product line evolution. If we consider the discussion so far, we can derive some requirements for such models of product line evolution:

The model should be able to support *evolution planning on feature level*. For instance, it should be possible to describe (i) the addition or deletion of features, (ii) structural changes (e.g., moving a feature) and (iii) changes to relationships between features.

In planning the evolution of product lines, it is also important that *partial decisions* and incomplete constraints are supported. Often, although the strategy for the next years is given to a certain degree, not all decisions are already defined. So it is only partially known how the feature tree of the next generation will look like. Hence, it should be possible to take the known facts into account while assuming that more changes are possible.

When planning future changes of product lines it is desirable to analyze and gather the *resulting consequences*, even if the underlying models are large and complex. In real life projects, feature models can become very complicated with large numbers of interrelated features. Hence, evolution decisions and their potential consequences should be presented with means that support cognition and increase usability, e.g., by providing interactive access to the relevant information and abstracting away unnecessary details.

Because of the overall complexity of evolution, it is necessary to support the engineer with automated analyses, which summarize information or detect relevant constellations of particular interest. For instance, violated constraints should be detected and resolved.

3. EVOFM: A FEATURE MODEL FOR EVOLUTION PLANNING

In this chapter we take first steps towards a solution which satisfies the requirements described earlier. In Section 3.1 we consider different solution alternatives and argue why we select a feature-driven approach. Then, in Section 3.2 we present the basic principles for *EvoFM*, a feature model for evolution planning. Finally, we illustrate these ideas by a first example in Section 3.3.

3.1 Analysis of Solution Alternatives

In our view of the problem, we assume that the planning of proactive evolution (as described above) is performed *on feature level*. This section discusses, how to provide an appropriate representation to handle the complexity.

We interpret the evolution of a product lines as a sequence of feature models – where each model is an evolution of the one before. As argued above, in real practice such models can become very complex. Moreover, they contain a high amount of information that is *not* affected by the evolution (i.e., elements that remain stable) and, hence, has less relevance for the planning. In addition, the evolution decisions are gathered incrementally over several planning sessions, such that initially it is impossible to define complete models for all relevant future steps. Hence, specifying the evolution just in terms of a sequence of feature models seems to be an insufficient solution.

An alternative is to focus only on the *differences* between models. For instance, starting from historic versions of the feature model (e.g., in year 2009 considering the versions from 2008 and 2009), the developers might define the changes planned for future versions (e.g., for 2010 and 2011).

This solution solves many of the problems above: The complexity is reduced, as only the changes have to be considered. These differences could be handled and represented, e.g., with concepts from model comparison. However, simple model comparison techniques only cover comparison between two or three models and, hence, are not sufficient for our purposes.

Due to these drawbacks, an alternative is required, which provides a better overview on the changes between multiple feature models. We argue that feature models themselves are such a tool as they provide support to specify configuration choices and variabilities on a certain abstraction level. Moreover, the developers in context of (feature-based) product lines are already familiar with feature models. Thus we argue, that it is a logical consequence to leverage feature models as a tool for product line evolution planning: Features can be used to show the commonalities and, in particular, the variabilities between the different evolution steps. Each evolution step then corresponds to a *feature configuration*, i.e., a concrete set of features describing the product line under evolution at a concrete point of time. These feature configurations can be visualized, e.g., in form of a feature matrix, to provide a compact and intuitive overview on the product line evolution.

3.2 Basic Principles

We use a basic feature model based on FODA [13] consisting of a hierarchy of features and cross-tree constraints between them. For the further discussion, we will call this feature model *EvoFM* (*Evolution Feature Model*). As elaborated above (Section 3.1), the main idea of EvoFM is to capture the commonalities and variabilities of multiple feature models. Consequently, EvoFM is based on the following principles:

1. A feature in EvoFM represents one or more feature model elements on product line level (e.g., a whole subtree). This supports abstraction.
2. The feature types and the constraints in EvoFM define the possible commonalities and variabilities during evolution. For instance, mandatory features in EvoFM represent parts which remain stable during evolution (i.e., are present in all product lines) while optional features represent variable parts (which are affected by the evolution).
3. A configuration of EvoFM corresponds to one evolution step, i.e., a concrete product line. The sequence of configurations (represented, e.g., in a configuration matrix) can be used for compact visualizations of product line evolution over time.

For simplification, we assume that identifiers of features remain consistent. In other words, we assume that during the evolution (1) no feature is renamed and (2) if a feature f is removed during evolution and a few evolution steps later a new feature f is added, we assume that this is the same feature, which is re-introduced into the product line.

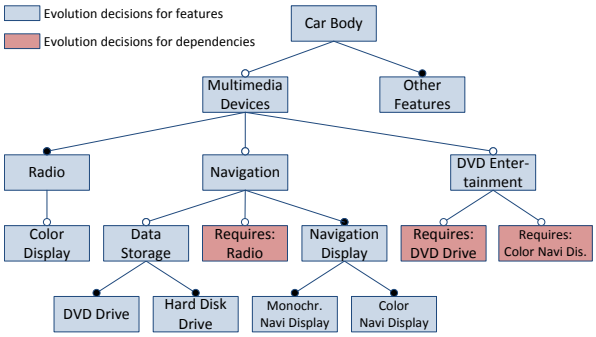


Figure 2: EvoFM for the automotive example.

	2008	2009	2010	2011
Multimedia Devices	X	X	X	X
Radio	X	X	X	X
Color Radio Display	X	X		X
Navigation		X	X	X
Navigation requires Radio		X		X
Data Storage				X
DVD Drive				X
Hard Disk Drive				X
Navigation Display			X	X
Monochrome Navi Display			X	X
Color Navi Display			X	X
DVD Entertainment				X
DVD Entertainment requires DVD Drive				X
DVD Entertainment requires Color Display				X

Figure 3: Configuration matrix for EvoFM.

3.3 Example of an EvoFM

This section shows a possible EvoFM for the automotive example presented earlier in Figure 1. Hence, we make concrete proposals how EvoFM could be realized in detail.

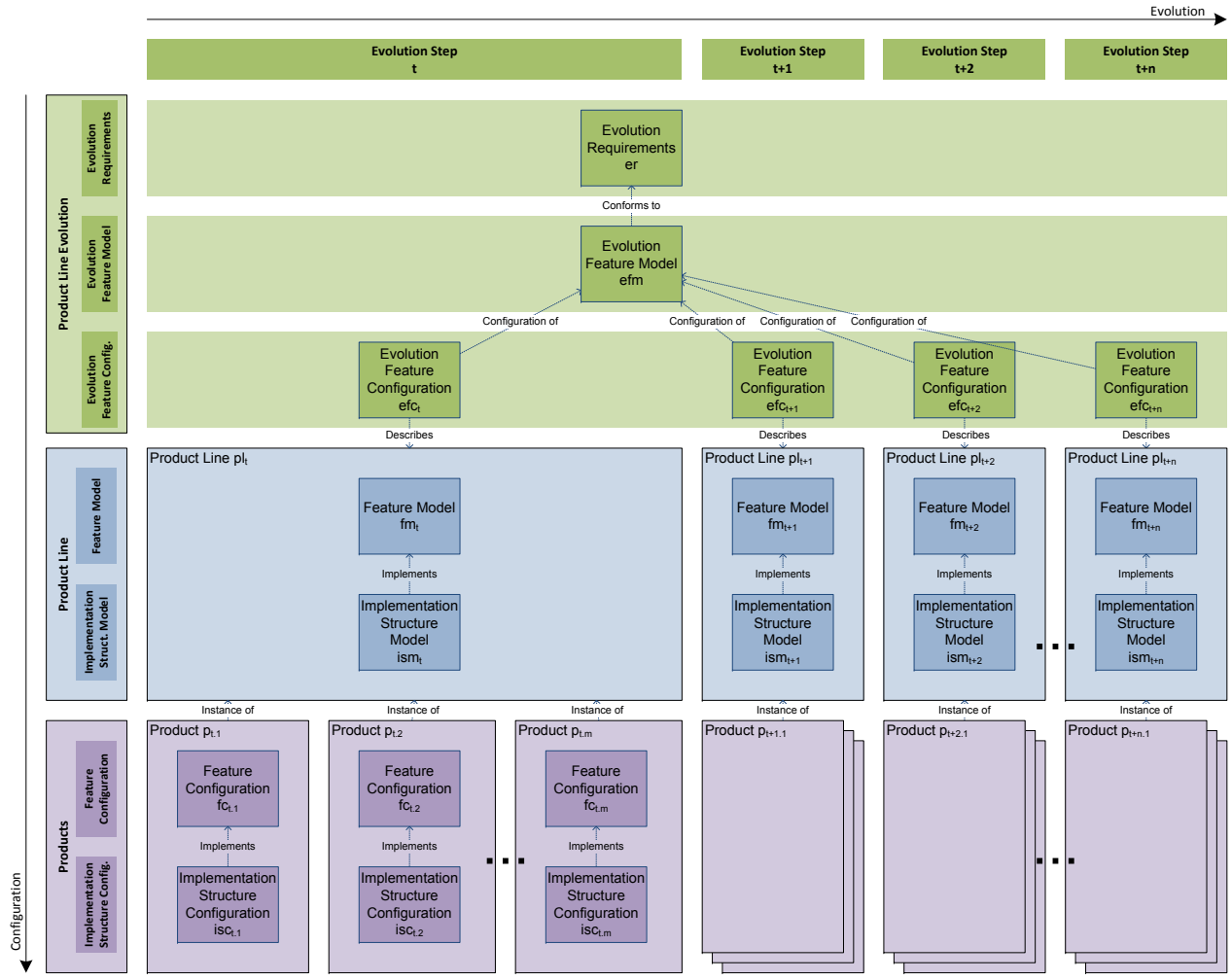


Figure 4: EvoSPL – Framework for the Evolution of Product Lines.

Figure 2 shows EvoFM for the automotive example from Figure 1. As described by the basic principles (Section 3.2), the product line features that vary over time are represented by optional features in EvoFM. For instance, *Navigation*, which has been introduced into the product line in 2009 (Figure 1), is represented in EvoFM by a corresponding optional feature (Figure 2). Features that remain stable are represented by mandatory features in EvoFM, e.g., *Radio*.

To reduce complexity, all subtrees (in the PL feature model) which remain stable are abstracted to a single (mandatory) feature in EvoFM, which is named like the root feature of the represented subtree. For instance, the feature *Monochrome Display* is not shown explicitly in EvoFM as, here in the example, it is not affected by the evolution and implicitly part of the subtree *Radio*. This means in turn, that *Radio* in EvoFM represents multiple features in the product line, namely *Radio* and *Monochrome Display*.

For cross-tree constraints we apply similar principles. Constraints that remain stable over time are hidden in EvoFM. However, if required it is possible to explicitly specify that a constraint varies over time. In that case, the ability to affect the constraint by evolution is represented in EvoFM as a special optional feature. For instance, in the product line in 2009, the *Navigation* requires *Radio* while in 2010 *Radio* and *Navigation* are independent. This is represented in EvoFM by an additional feature *Requires: Radio* specified as subfeature of *Navigation*.

Figure 3 shows the configuration matrix of EvoFM, which provides a compact overview on the product line evolution. Each column corresponds to a feature model of the product line (cf. Figure 1).

4. EVOSPL: A FRAMEWORK FOR THE EVOLUTION OF PRODUCT LINES

The further discussion will proceed in two steps: In this section, we will define *EvoSPL*, a framework for the evolution of product lines. In particular, this framework defines the various models that are involved in the evolution process and, hence, provides context for EvoFM introduced earlier. In the next section we will then define several basic scenarios that apply the framework.

The EvoSPL framework (see Figure 4) is organized along two dimensions. The horizontal dimension shows the **Evolution**, starting with the *Evolution Step* t over several steps $t+1, t+2, \ldots, t+n$ and so forth. Depending on the usage of the framework, along this axis we might consider the current situation, historic evolution steps in the past or planned future evolution steps.

Orthogonal to the evolution we distinguish three levels of **Configuration**. From top to bottom these are:

Product Line Evolution – On the highest abstraction level we have artefacts that describe the product line evolution. *Evolution Requirements* er give guidelines and set constraints for the further planning of evolution. The *Evolution Feature Model (EvoFM)* efm has to conform to these constraints and describes the evolutionary changes between *Evolution Steps* in terms of features. For each *Evolution Step* i there is an *Evolution Feature Configuration* efc_i, which describes the *Product Line* pl_i in terms of feature configuration decisions of the *Evolution Feature Model (EvoFM)* efm.

Product Line – Each of these evolution steps i is a product line pl_i consisting of a *Feature Model* fm_i and an *Implementation Structure Model* ism_i. To support this correspondence between the EvoFM and the various evolution steps, the EvoFM is mapped onto product line models by separate mapping models. To simplify the illustration, these have been omitted from Figure 4.

Products – The third and most concrete level of configuration is given by products. For the first evolution step, i.e., step t, Figure 4 shows several products, which have been created as instances of the product line pl_t. These products are described by *Feature Configurations* $fc_{t.1}, \ldots, fc_{t.m}$ (configurations of fm_t) and corresponding *Implementation Structure Configurations* $isc_{t.1}, \ldots, isc_{t.m}$ (configurations of ism_t), which implement these feature configurations. For the subsequent evolutions steps $(t+1, \ldots, t+n)$ these product artefacts have been omitted to simplify the illustration.

5. SCENARIOS

This section describes three different scenarios for the use of our framework. Scenario 1, *Reactive Derivation* (Section 5.1), describes how an EvoFM can be derived from an existing sequence of feature models that describes the product line evolution. Scenario 2, *Proactive Planning* (Section 5.2), describes how to create an EvoFM from high-level evolution requirements. Scenario 3, *Analysis* (Section 5.3), discusses how an existing EvoFM can be used for analyzing planned future evolution steps.

5.1 Reactive Derivation of EvoFM

In *Reactive Derivation* (see Figure 5) we assume that there is already information given about the evolution in terms of a sequence of feature models $(fm_t, \ldots, fm_{t+n})$. These feature models could either reflect historic data on previous evolution or result from planning of future evolution or both (i.e., $t \leq t_{current} \leq t+n$).

Given the sequence of feature models, one can derive

- a corresponding Evolution Feature Model (EvoFM) efm, which summarizes this evolution path,
- $n+1$ Evolution Feature Configurations $(efc_t, \ldots, efc_{t+n})$ describing the n evolution steps as configurations of the EvoFM, and
- mappings from EvoFM features to elements in the product line feature models.

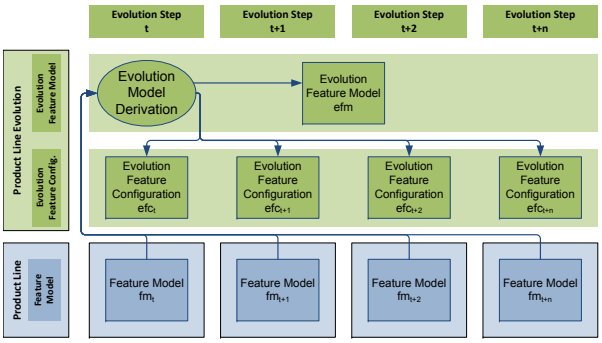


Figure 5: Scenario: Reactive derivation.

Metamodel Element	Change (Diff)	Change Interpretation	Represented in EvoFM as (i.e., maps to)
Class *Feature*	Add, Delete	Add/delete feature	Optional feature
	Modify	Move feature, e.g., restructure feature groups	*See discussion in text*
Attribute *name* (of Feature)	Modify	Replace a feature	Two features, one with the old name and one with the new name, as XOR (as subfeatures of feature representing this feature)
Attributes *min*, *max* (of Feature)	Modify	Define mandatory feature as optional or vice versa	Mandatory feature "mandatory"/"optional" (as subfeatures of feature representing this feature)
Class *FeatureGroup*	*Analogous to Feature*		
Attributes *min*, *max* (of FeatureGroup)	Modify	Redefine XOR group as OR or vice versa	Optional feature "XOR"/"OR" (as subfeature of feature representing this feature group)
Class *Dependency*	Add, Delete, Modify	Add/delete cross-tree constraint	Optional feature for the constraint (as subfeature of feature representing the source feature)
Reference *source*, *target* (of Dependency)	Add, Delete, Modify	Replace cross-tree constraint by another one	Two features, one for old constraint and one for new constraint, as XOR (as subfeatures of feature representing the source feature)

Table 1: Transformation concepts for mapping common changes in feature models to EvoFM elements.

We will now take first steps towards an automatic derivation of EvoFM. The resulting generated EvoFM can be manually refined and be used for further planning, like the definition of additional evolution steps in terms of EvoFM configurations.

The automatic derivation as illustrated in Figure 5 is performed in three steps: First, we perform a model comparison between the product line feature models to calculate the commonalities and differences between them and store them as a model (*diff model*). Second, we use a model transformation on the resulting diff model to create both the EvoFM and the mapping from EvoFM to product line feature models. In a third step, the configurations for the existing evolution steps can be calculated by comparing each product line feature model with the EvoFM.

For the first step, we use *EMF Compare* [11] for model comparison. EMF Compare calculates a model of the commonalities (*match model*) and a model of the differences between compared models (*diff model*). (More semantically rich comparisons are possible with other techniques (e.g.,

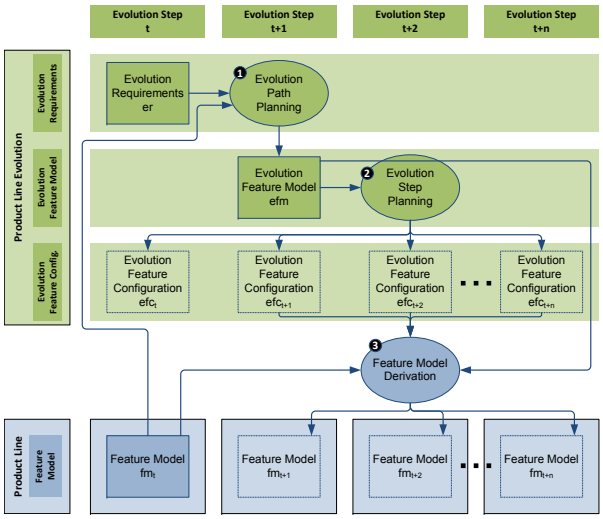


Figure 6: Scenario: Proactive planning.

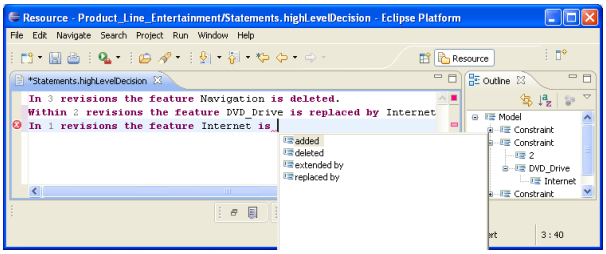


Figure 7: DSL editor for evolution requirements.

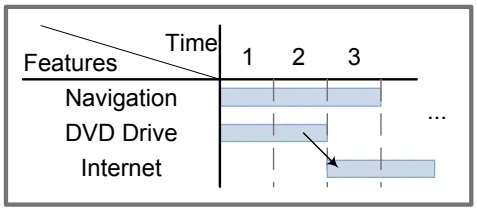


Figure 8: Roadmap view for evolution requirements.

[14]). Our prototype is currently restricted to a set of two feature models (EMF Compare supports comparison of up to three models).

Extension towards multiple feature models would require to perform multiple comparisons, e.g., against the first feature model in the evolution sequence.

For the second step, we use a model transformation written in ATL [9]. It takes the feature models and the diff model resulting from the first step as input. To define the transformation rules creating EvoFM, all possible changes during the evolution of a feature model have to be considered. On technical level, EMF Compare distinguishes between (1) addition, (2) deletion, or (3) modification of (a) model elements (b) attributes, or (c) references (see [1] for a general discussion).

Table 1 provides an overview on relevant feature model elements, possible changes on them, and their mapping to EvoFM elements. The first column shows the model elements from the feature meta-model which have to be considered. The second column ("Change") shows the types of changes to be considered for each model element. The third column ("Change Interpretation") describes how these changes are interpreted on conceptual level. The last column shows the corresponding EvoFM elements, where each type of change is mapped to. The table rows correspond to (summarized) transformation rules for the automatic derivation of an EvoFM like the one shown in Section 3.3. These rules indirectly define the semantics of EvoFM.

We would like to discuss one specific case from the table in greater detail: the moving of a feature within the hierarchy. This kind of change often occurs when modelers (re-)structure features into feature groups. An example can be found in Figure 1: In 2010, *Monochrome Navi Display* and *Color Navi Display* are direct children of *Navigation* while in 2011 they are subfeatures of *Navigation Display*. Currently, we handle this issue by just including *Navigation Display* into EvoFM. A possible future solution could be to introduce a cross-tree constraint for EvoFM which allows to define that *Navigation Display* is only available in the product line if *DataStorage* is available as well.

5.2 Proactive Planning of Evolution

In this section we consider how long-term decisions made by upper management can be transformed into an EvoFM. Since the decisions are done in a business context they can differ from technical requirements. We assume that we have a scenario as shown in Figure 6. The feature model fm_t and the evolution requirement er (representing constraints set by management) are given and we want to derive the evolution feature model efm and the feature models fm_{t+1}, fm_{t+2} and so on.

The evolution requirements er represent constraints given by a manager like: (1) *The DVD Drive will be replaced by an internet streaming connection within the next three revisions*, (2) *The navigation will be removed within the next four revisions*, or (3) *An internet connection will be introduced in the next revision.*

To gather the requirements in a semi-formal form we introduce a simple domain-specific language (DSL) which allows to express constraints as textual statements. We use Xtext [10] to define the grammar and derive corresponding tools, e.g., a language-aware editor with syntax highlighting and auto-completion (see Figure 7). The information specific in this language can be used to derive graphical roadmap views similar to the one sketched in Figure 8.

We are currently experimenting with model transformations that read this DSL document representing the manager's statements and the current feature model fm to produce a preliminary version of the evolution feature model (EvoFM) efm (cf. the transformation ❶ in Figure 6). To this end, we use the feature model fm to identify the structure of features and the evolution requirements er to define and extract changes. In contrast to the automatic derivation in Section 5.1, here we do not know *where* new features should be located in the evolving feature model. We are currently experimenting with two options: Either new features are inserted as children of a special *Unsorted*-feature or the developer can augment the er specification with a desired

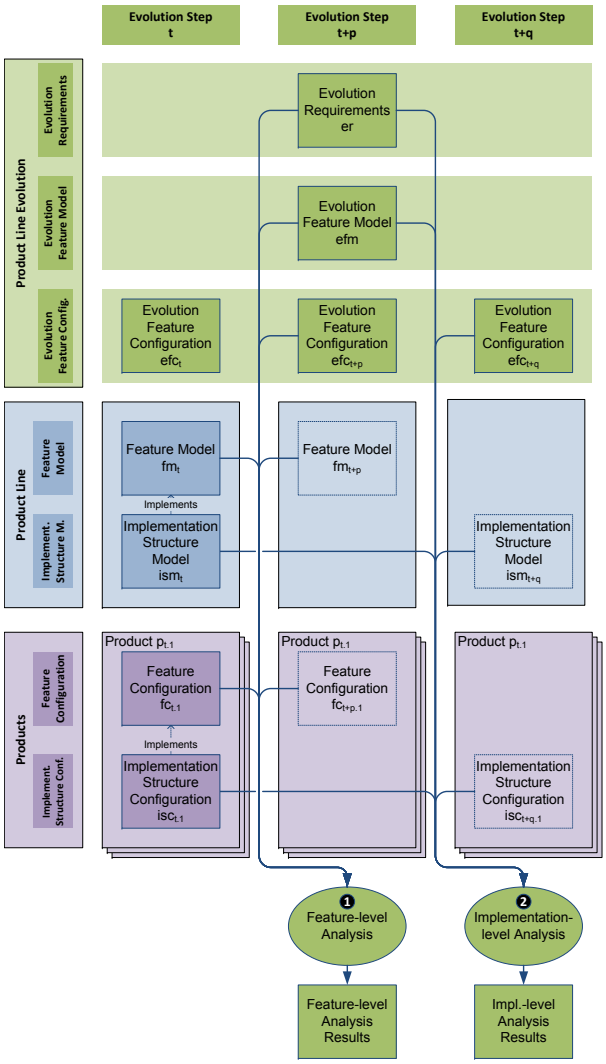


Figure 9: Scenario: Analysis.

location ("introduce x as subfeature of y").

In a second step (❷ in Figure 6) we use the EvoFM to derive the possible evolution feature configurations efc_{t+1}. Finally, in a third process (❸ in Figure 6) we can use the mapping between the EvoFM efm, the feature model fm_t, and the evolution feature configuration efc_x to generate the feature models fm_{t+x-1}.

5.3 Analysis with EvoFM

When analyzing a planned evolution of a product line we want to answer questions like (1) *Are the changes and evolution decisions consistent?* (2) *Is it still possible to provide all existing products?* (3) *Which features have to be implemented or redesigned?*

(1) In general the Feature Model fm_t contains relations which express technical or business requirements which have to be fulfilled. When evolving, we can end up with an new model fm_{t+p} which is inconsistent. This can be checked with various automated analyses (e.g., by translation into logic and checking for satisfiability).

(2) Given some existing products we might ask whether all of these can still be provided using the evolved Feature Model fm_{t+p}. To answer this, we can check existing Feature Configurations $fc_{t,x}$ against the new Feature Model fm_{t+p} (e.g., again by checking satisfiability).

(3) Beside these feature-level analyses (❶ in Figure 9), the framework also allows implementation-level analyses (❷ in Figure 9). These could provide additional information, e.g, costs and consequences (in the implementation) for planned evolution steps.

In general we want to increase the reusability of components and support "evolution-aware" architectural decisions. For instance, feature-implementation mappings allow to determine features which currently have no implementation in the Implementation Structure Model ism_{t+q}. Similarly, with additional information about the cost of implementing a feature, provided, e.g., by experts, we can estimate the cost of the planned evolution.

In order to support architectural decisions, additional information for the developer is required. Using EvoFM we intend to provide different task-specific views, e.g., to clarify the differences between Feature Models fm_t and fm_{t+p} and to help to identify whether a component is stable or not. Moreover, during design and construction of implementation components, we can provide additional information, e.g., that a particular component has to be modified next year and becomes obsolete one year later.

6. RELATED WORK

Several existing approaches deal with product line evolution on different levels of abstraction. According to [6], one can distinguish between product-specific adaptation, reactive evolution, and proactive evolution. In this paper we focus on proactive evolution, i.e., active planning of future versions on domain level. [19] shows the different dimensions of evolution and handles them in terms of an evolution graph. From this point of view, our paper focuses on the product line dimension, while the evolution (maintenance) of concrete products is not considered. Two concrete case studies on product line evolution and a classification of possible changes during evolution can be found in [21].

Most of the existing work deals with the implementation issues for evolving product lines like [15], which presents an approach based on the combination of aspect-oriented programming and frames. Others, like [16, 7, 8] use concepts from model-driven development. For instance, [7] addresses the domain evolution of model-based product lines by the example of embedded systems. The authors present an approach to provide meta-model-based transformations to support systematic evolution steps.

7. CONCLUSIONS

We discussed first steps towards proactive planning of product line evolution. Our research motivated by observations from industrial practice, in particular from the area of embedded systems. In such real life applications of SPL, there is a need for long-term planning of product portfolios on feature-level. However, while several approaches address evolution in the implementation, support on a more abstract level that facilitates systematic planning and handling of

complex evolution scenarios is still missing.

Based on an initial analysis (Section 3.1) we argue that feature models seem to be a promising tool to specify, manage, and analyze product line evolution more systematically and on an appropriate level of abstraction. To this end, we propose EvoFM, a feature model for product line evolution (Section 3) to capture commonalities and variabilities between multiple evolution steps. Consequently, each evolution step can be represented as a EvoFM configuration, which can be leveraged as a base for compact high-level representations and visual tools. Furthermore, we describe a corresponding framework for feature-driven evolution planning (Section 4). On this base we show three typical application scenarios and discuss corresponding tool support (Section 5).

Future work will include investigations towards combination with other approaches for evolution planning. For instance, the area of *Technology Roadmapping* [18] provides concepts to visualize different aspects of product planning (like business objectives, markets, products, technologies, milestones, etc.) and the relationships between them – similar to the sketch in figure 8.

On the lower level, combination with existing approaches for model-driven evolution on implementation level (as described in section 6) seems promising. In the long run, this could lead to an overall model-driven framework, which allows to consistently plan, analyze, and deploy product line evolution on all levels of abstraction with seamless transitions between them.

Altogether, we believe that feature-driven evolution planning – as introduced here in a very first step – might open up several new research opportunities towards a systematic feature-oriented engineering of software evolution.

8. REFERENCES

[1] M. Alanen and I. Porres. Difference and union of models. In *The Unified Modeling Language, Modeling Languages and Applications, 6th International Conference (UML 2003)*, pages 2–17, San Francisco, CA, USA, October 2003.

[2] S. Apel, C. Kastner, and C. Lengauer. Featurehouse: Language-independent, automated software composition. In *31st International Conference on Software Engineering (ICSE '09)*, pages 221–231, Washington, DC, USA, 2009. IEEE Computer Society.

[3] D. Batory, J. Sarvela, and A. Rauschmayer. Scaling step-wise refinement. *IEEE Transactions on Software Engineering*, 30:1278–1295, 2004.

[4] P. Clements and L. M. Northrop. *Software Product Lines: Practices and Patterns*. The SEI series in software engineering. Addison-Wesley, Boston, MA, USA, 2002.

[5] K. Czarnecki and U. W. Eisenecker. *Generative Programming*. Addison Wesley, Reading, MA, USA, 2000.

[6] S. Deelstra, M. Sinnema, and J. Bosch. Product derivation in software product families: a case study. *Journal of Systems and Software*, 74(2):173–194, 2005.

[7] G. Deng, G. Lenz, and D. C. Schmidt. Addressing domain evolution challenges in software product lines. In J.-M. Bruel, editor, *MoDELS Satellite Events*, volume 3844 of *LNCS*, pages 247–261. Springer, 2005.

[8] D. Dhungana, T. Neumayer, P. Grunbacher, and R. Rabiser. Supporting evolution in model-based product line engineering. In *12th International Conference on Software Product Lines (SPLC 2008)*, pages 319–328, Limerick, Ireland, September 2008. IEEE Computer Society.

[9] Eclipse-Foundation. ATL (ATLAS Transformation Language). http://www.eclipse.org/m2m/atl/.

[10] Eclipse-Foundation. Xtext. http://www.eclipse.org/Xtext.

[11] Eclipse Modeling Framework Technology (EMFT). EMF Compare. http://wiki.eclipse.org/index.php/EMF_Compare.

[12] C. Kaestner, T. Thum, G. Saake, J. Feigenspan, T. Leich, F. Wielgorz, and S. Apel. FeatureIDE: A tool framework for feature-oriented software development. In *ICSE '09: Proceedings of the 2009 IEEE 31st International Conference on Software Engineering*, pages 611–614, Washington, DC, USA, 2009. IEEE Computer Society.

[13] K. Kang, S. Cohen, J. Hess, W. Novak, and S. Peterson. Feature oriented domain analysis (FODA) feasibility study. SEI Technical Report CMU/SEI-90-TR-21, ADA 235785, Software Engineering Institute, 1990.

[14] D. S. Kolovos. Establishing correspondences between models with the epsilon comparison language. In *5th European Conference on Model Driven Architecture - Foundations and Applications*, pages 146–157, Enschede, The Netherlands, 2009.

[15] N. Loughran, A. Rashid, W. Zhang, and S. Jarzabek. Supporting product line evolution with framed aspects. In *AOSD ACP4IS Workshop*, 2004.

[16] T. Mens and T. D'Hondt. Automating support for software evolution in UML. *Automated Software Engineering*, 7(1):39–59, 2000.

[17] D. Parnas. On the design and development of program families. *IEEE Transactions on Software Engineering*, SE-2(1):1–9, March 1976.

[18] R. Phaal, C. J. P. Farrukh, and D. R. Probert. Technology roadmapping–a planning framework for evolution and revolution. *Technological Forecasting and Social Change*, 71(1-2):5 – 26, 2004.

[19] S. Schach and A. Tomer. Development/maintenance/reuse: software evolution in product lines. In *1st Software Product Lines Conference (SPLC 2000)*, pages 437–450, Denver, Colorado, August 28-31 2000.

[20] Software Engineering Institute. SPL Hall of Fame. Web site, 2008. http://splc.net/fame.html.

[21] M. Svahnberg and J. Bosch. Evolution in software product lines: Two cases. *Journal of Software Maintenance: Research and Practice*, 11(6):391–422, 1999.

[22] M. Voelter and I. Groher. Product line implementation using aspect-oriented and model-driven software development. In *11th International Software Product Line Conference (SPLC 2007)*, Kyoto, Japan, September 2007.

[23] D. M. Weiss and C. T. R. Lai. *Software Product Line Engineering: A Family-Based Software Development Process*. Addison-Wesley, 1999.

Detecting Feature Interactions in SPL Requirements Analysis Models

Mauricio Alférez, Ana Moreira
Universidade Nova de Lisboa
Caparica, Portugal
{mauricio.alferez, amm}@di.fct.unl.pt

Uirá Kulesza
UFRN
Natal, Brazil
uira@dimap.ufrn.br

João Araújo, Ricardo Mateus, Vasco Amaral
Universidade Nova de Lisboa
Caparica, Portugal
{ja, vasco.amaral}@di.fct.unl.pt
rjm17469@fct.unl.pt

ABSTRACT
The consequences of unwanted feature interactions in a Software Product Line (SPL) can range from minor problems to critical software failures. However, detecting feature interactions in reasonably complex model-based SPLs is a non-trivial task. This is due to the often large number of interdependent models that describe the SPL features and the lack of support for analyzing the relationships inside those models. We believe that the early detection of the points, where two or more features interact —based on the models that describe the behavior of the features —, is a starting point for the detection of conflicts and inconsistencies between features, and therefore, take an early corrective action.

This vision paper foresees a process to find an initial set of points where it is likely to find potential feature interactions in model-based SPL requirements, by detecting: (i) dependency patterns between features using use case models; and (ii) overlapping between use case scenarios modeled using activity models.

We focus on requirements models, which are special, since they do not contain many details about the structural components and the interactions between the higher-level abstraction modules of the system. Therefore, use cases and activity models are the means that help us to analyze the functionality of a complex system looking at it from a high level end-user view to anticipate the places where there are potential feature interactions. We illustrate the approach with a home automation SPL and then discuss about its applicability.

Categories and Subject Descriptors
D.2.1 [**Software Engineering**]: Requirements/Specifications
D.2.13 [**Software Engineering**]: Reusable Software – *domain engineering*

General Terms
Design, Verification.

FOSD'09, October 6, 2009, Denver, Colorado, USA.

Keywords
Software Product Lines Requirements, Feature Interactions.

1. INTRODUCTION
In Software Product Line Engineering (SPLE) detecting potential unwanted interactions between features of an SPL is essential to produce correct products [1], i.e., a product whose features are not conflicting. In this paper, we use the definition of "feature interaction" given in [2], which states that a feature interaction exists when:

- "A service feature inhibits or subverts the expected behavior of another service feature".
- "The joint accurate execution of the functionality of two or more features provokes a supplementary phenomenon which cannot happen during the processing of each feature functionality, when considered separately".

The above definition of feature interaction focuses on the manifestation of the interaction, and is applicable where unexpected and undesired behavior occurs. FIs could be analyzed in Domain Engineering to anticipate unwanted interactions between SPL features, and also in Application Engineering during the selection of features for specific products. In the literature, the terms "feature interference" or "bad feature interaction" are synonymous to refer to an undesired interaction. We will simply use the expression "feature interactions" (FIs) when referring to unwanted interactions.

Recent research on SPL Engineering (SPLE) aims at finding ways to detect, as well as to resolve feature interactions. Some of these approaches rely on manual creation of detailed models that formalize the relationships between the features and allow the analysis and detection of feature interactions [3-4]. These approaches are useful to analyze the semantics of the interactions; this is significant when the modeler has an initial idea on the potential features that could present feature interactions. However, the creation of extra models to find FIs can become an arduous task if there is no clue about which are the set of features that intervene in a potential FI.

In other cases, there is support to automate the detection of FIs, but these are mainly focused on structural feature interactions during the composition of the models for a specific product of the SPL [5-7]. An example of this kind of FIs is when the design of a feature requires elements in the models that are only introduced by another feature that is not yet included in the system. In many

of these structural FIs, changing the order of the composition resolves the interaction. However, this approach does not address other kinds of FIs based on the meaning of the relationships between the models of features.

This paper aims to tackle the above mentioned difficulties. In particular, it addresses the need of a process to find an initial set of points in the models of features where it is likely to find potential FIs. The process is based on the semantics of the relationships between the elements in the requirements analysis models, and also their relationships with the features in the feature model. The advantages of using this kind of process are: i) it does not require other types of models (e.g., problems frames), ii) it only uses recurring models like use cases and activity models; and iii) it allows analyzing the functionality of a complex system looking at it from a high level end-user view using requirements models.

Specifically, our approach exploits the following two strategies:

1. *Dependency patterns*: These patterns take into consideration the semantics of the relationships between the elements in the requirements models to expose dependencies between the features. This information helps the developer to focus on the set of features that might be involved in FIs.
2. *Overlapping detection*: All variants of a product line may have models that describe them. Since the variants have usually significant overlaps in their functionality, their models have significant overlaps too [8]. The identification of the points of overlapping between models of features can assist the developer to focus on the places where features might be interacting in specific use case scenarios.

The remaining of this paper describes in more detail these two strategies (Section 2), and then it illustrates their application in a home automation case study (Section 3). It follows by discussing the benefits and applicability of these strategies (Section 4). Finally, it concludes and outlines future work (Section 5).

2. DETECTING POINTS OF INTERACTION

We propose two strategies to detect points of interaction between features in SPL requirements. The first strategy, dependency pattern detection, is used when the source requirements models handle coarse-grained information. The second strategy, overlapping detection strategy, complements the first, and is primarily used when interactions can be detected in behavioral models containing fine-grained information. Next we sketch the application process of these strategies and detail each one.

2.1 Process Overview

The use of the two strategies to identify candidate points of interaction between features assumes a process workflow, described in Figure 1.

This is a process where the inputs are: the use case model of the SPL (a); the feature model (b); and the relationships, also called trace links, between the features in the feature model and the elements in the use case model (c).

Dependency patterns analysis (d) is based on the study of the "include" and "extend" relationships in use case models. It is applied to discover possible hidden dependencies between features that might indicate points of potential FIs. The result (e) is used as a guide to identify which use cases (related with the dependent features) have to be specified using activity models. Activity models are used to model the scenarios in which the features may interact. The modeler can use existent models (f) or provide new or updated versions (g-h). Then, he/she should provide the updated trace links between features and model elements of the activity models such as activity models, activities and activity partitions (i).

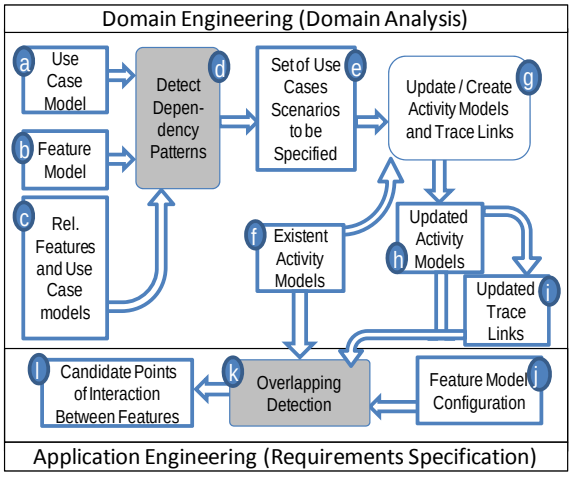


Figure 1. Overview of the Process Workflow

The updated trace links (i) that relate the elements in the activity models (f-h), and the features in a specific product of the product line (j), are used to detect overlapping (k) between the use case scenarios related with the use cases identified in (d). Finally, "overlapping detection" presents the candidate points of interaction between features manifested in specific activities of the activity models (l). In this paper we focus on the two steps on Figure 1 highlighted in grey.

2.2 Detect Dependency Patterns

Some dependency patterns suggest where a potential feature interaction is likely to be found. Also, they can suggest where more detail is required about the behavior of a feature to identify the presence of a feature interaction. Such patterns can be recognized in coarse-grained requirements models elements like UML2.0 use cases and their relationships.

A "dependency", in the context of use case modeling, indicates that a change in one use case can affect another use case. Our hypothesis is that the analysis of the dependencies between use cases can help to detect dependencies between the features that they model. In this work we explore two well-known use cases relationships: "includes" and "extends".

In UML modeling, the "include" relationship states that one use case (the base use case) includes the functionality, or behavior, defined in another use case (the inclusion use case). The "include" relationship supports the reuse of functionality in a use case model. Therefore, the developer can add "include" relationships to a use case model to show the situation in which the behavior of the inclusion use case is common to two or more use cases.

The "extend" relationships can be used to specify that one use case (extension) extends the behavior of another use case (base). The extend relationship specifies that the incorporation of the

extension use case is dependent on what happens when the base use case executes. This type of relationship reveals details about a system or application that are typically hidden in a use case. It is used in situations when: (i) part of a use case is optional system behavior; (ii) a sub flow is executed only under certain conditions; and (iii) a set of behavior segments may be inserted in a base use case.

The "includes" and "extends" relationships imply a dependency between base and inclusion use case, and base and extension use case. Figure 2 provides a catalog of typical patterns where features are related with use cases. We consider these relationships "trace links" between the SPL features modeled in taxonomic view provided by the feature model, and their semantics provided by the use case and activity models.

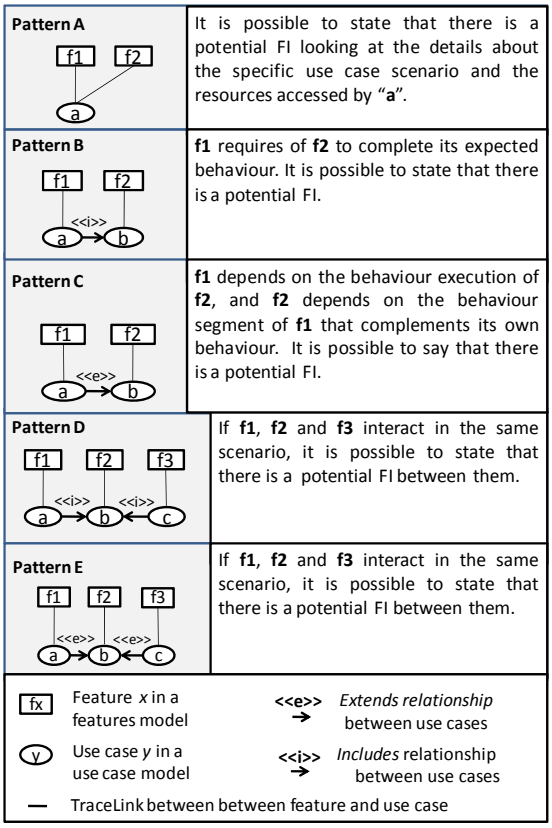


Figure 2. Patterns Catalog that Provide Information to Detect Potential Feature Interactions

It is important to note that these dependency patterns are useful to identify places (e.g., specific use cases involved in the patterns), and set of features involved in potential feature interactions. Providing details about the use cases' scenarios gives more information for the modeler to determine if the potential feature interactions are in fact alerting about a real feature interactions. Once use case scenarios are specified, the strategy "Overlapping Detection" could be applied to support the search of more specific points where the features interact.

The idea of detecting dependencies as a base to find points of interaction between features is not completely new. Metzer [1] suggested an approach based on specifying detection concepts at the model level. The feature interactions are detected at different levels, but are generally considered to be dependencies between functional needs. Our contribution is to provide a specific catalog of patterns in UML Use Case models, taking into account specific types of relationships to infer functional dependencies between the features.

2.3 Overlapping Detection Strategy

We use UML use case and activity models to model the requirements behavior of each feature. We aim to find the specific model elements in the whole set of the activity models, in which the behavior of one feature influences the behavior expressed in the models of other features.

Figure 3 sketches the relationships between the main elements involved in a process of identifying the points of interaction between features. It shows a feature model configuration for the specific product "H" in the product line (top layer), the relationships or trace links between features and requirements represented as a tuple <traceLinkId, FeatureId, ModelId, ElementId> (middle layer), and requirements models (bottom layer). Each feature (e.g., A, B, and C) is related to model elements in the requirements models through trace links. The meaning of each trace link is that the model element describes part of the intended behavior of the feature.

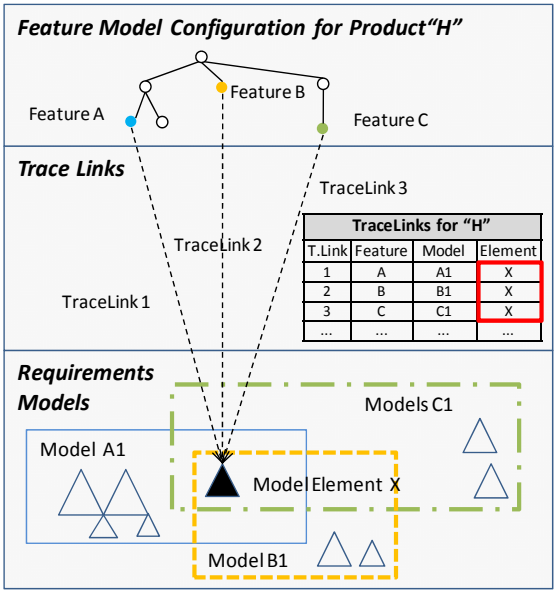


Figure 3. Feature Interaction Detection based on the Overlapping Strategy

The base strategy to find feature interactions between features A, B and C is to find the overlapping elements of their behavioral models in specific use case scenarios. The model element X (bottom layer) in the requirements models layer represents a triple overlap, where this model element, shared between the set of

requirements models, has trace links with features A, B and C of the feature model configuration (top layer).

3. ILLUSTRATIVE EXAMPLE

We use the Smart Home SPL to illustrate our feature interaction detection process sketched in Figure 1. This case study was provided in the context of the European AMPLE Project [9] by our industrial partner Siemens A.G [10-11].

Figure 4 shows a simplified Smart Home feature model and one of its configurations called Economic Smart Home where the features included for the specific product are ticked and the ones that are excluded are marked with a cross.

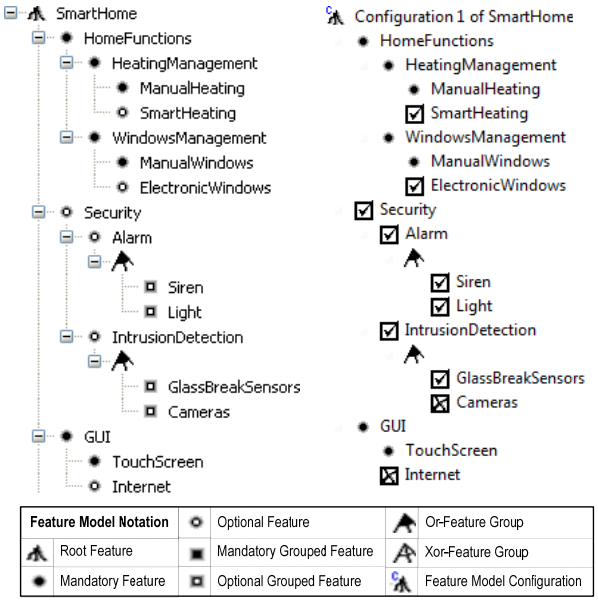


Figure 4. Smart Home SPL Feature Model (Left); the Economic Smart Home Configuration (Right)

Figure 5 shows the use case model for the economic version of the Smart Home. In this model we marked in bold the model elements that are not common to all the products of the Smart Home SPL, but were included in the Economic version.

Table 1 shows part of the relationships between some of the features in Figure 4 and requirements model elements in Figure 5 and Figures 7–9. The extra column "model elements container" eases the identification of each model element in the models.

The process described in this paper assumed the existence of links between features and other kind of models like the ones shown in Table 1. These links can be created manually, or semi-automatically. In a previous work we addressed the semi-automatic creation of links using the VML4RE language [12]. This allows the creation of trace links programmatically in a separate specification using designators and quantification. Also it provides transformation operators specially tailored to requirements analysis models like use cases, activity and goal models. These operators ease the customization of the SPL models for specific products such as the one shown for the economic home in Figure 5.

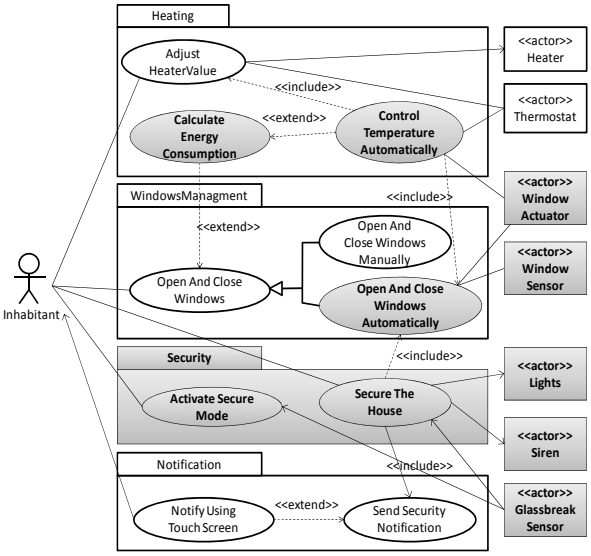


Figure 5. Economic Smart Home Use case Model

Table 1. Trace Links Excerpt for Smart Home

Feature	Model Element		Model Element Container	
	Name	Type	Name	Type
Security	SecureTheHouse	UseCase	Security	Package
	SecureTheHouse	Activity Model	Ams	Activity Models
	Send Security Notification	UseCase	Notification	Package
	Windows Actuator	Activity Partition	SecureThe House	Activity Model
	Close Windows	Activity	Windows Actuator	Activity Partition
	OpenAndClose Windows Automatically	UseCase	Windows Management	Package
	…	…	…	…
	Control Temperature Automatically	UseCase	Heating	Package
Smart Heating	Smart Heating	Activity Model	Ams	Activity Models
	Windows Actuator	Activity Partition	SmartHeating	Activity Model
	Open Windows	Activity	Windows Actuator	Activity Partition
	Close Windows	Activity	Windows Actuator	Activity Partition
	OpenAndClose Windows Automatically	UseCase	Windows Management	Package
	Calculate Energy Consumption	UseCase	Heating	Package
	…	…	…	…
	OpenAndClose Windows	Activity Model	ams	Activity Models
Electronic Windows	Windows Actuator	Activity Partition	OpenAnd Close Windows	Activity Model
	Open Windows	Activity	Windows Actuator	Activity Partition
	Close Windows	Activity	Windows Actuator	Activity Partition
	OpenAndClose Windows Automatically	UseCase	Windows Management	Package
	…	…	…	…

Let us consider three features of the Smart Home case study: “Smart Heating”, “ElectronicWindows” and “Security”. Smart Home is designed to spend as less energy as possible. Therefore, each time the system has to change the indoor temperature it will first consider opening or closing windows, instead of using the heater. The respective features are “SmartHeating” and “ElectronicWindows”. Now, let us consider the “Security” feature. This feature implies that if one of the intrusion detection sensors is fired, the system starts to secure the house. This includes closing the windows, activating the alarm and sending a security notification to the inhabitants. Additionally, for non-economic Smart houses, the system will send an extra security notification to the security company via internet.

Imagine a situation where, during a hot summer day, the Smart Home decides to save energy and also to cool the house opening the windows: the house orders the windows actuator to open and keep the windows opened until reaching the right indoor temperature. Before the house reaches the right indoor temperature, one of the intrusion detection sensors is fired and the security process commands the windows to close. It is clear that the house should do first what the security process commands. Therefore, the developer had to describe what the Smart Home system should do when the electronic windows, smart heating, and security features interact in the same scenario in a specific moment. We will see how to systematically identify the places where such interactions may happen.

3.1 Applying Detection of Dependency Patterns

We illustrate in Figure 6 each pattern described in Section 2.2. We use as reference models the use case model of the Smart Home case study and some of the trace links of Table 1.

Pattern A: It is necessary to create more detailed behavioral models for the features “Smart Heating” and “Security” to determine if there is a real FI between them. Detailed models are used to see if these features are linked to the use case “Open and Close Windows Automatically” in the same use case scenario and if they share resources while opening and closing windows automatically. For example, there might be a feature interaction between “Security” and “Smart Heating” when requiring the execution of opposite commands like “open” and “close” at the same time over the shared resource “WindowsActuator”.

Pattern B: “Security” depends on the functionality provided by “Notification”. However, there is no evidence of a feature interaction between “Security” and “Notification” when sending a notification. It is necessary to model the internal behavior of each feature (Security and Windows Management) to see if there is information about behavior of one feature that subverts or inhibits the behavior of the other.

Pattern D: There is a potential feature interaction between “SmartHeating” and “Security” when opening and closing windows automatically. The use cases “ControlTemperatureAutomatically” and “SecureTheHouse” share a common behavior. The scenarios in which this common behavior appears, have to be modeled. This is necessary to design the possible corrective or preventive actions facing the interaction between the features at a specific time.

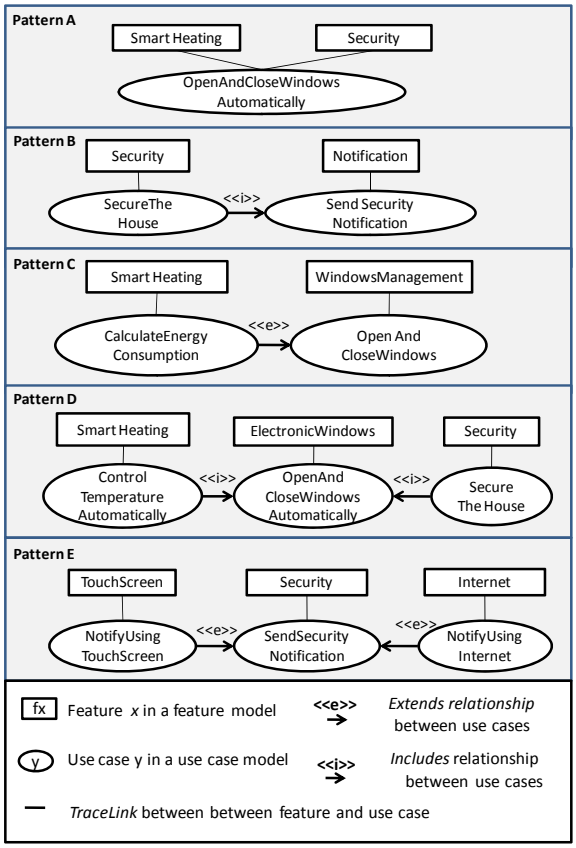


Figure 6. Patterns to Detect Candidate Feature Interactions in the Smart Home Case Study

Pattern E: There is a potential feature interaction between “TouchScreen” and “Internet”. This case does not correspond to the Economic version of the Smart Home because it does not include the optional feature “Internet”. However, we use it to illustrate the pattern E where there is a potential feature interaction between the two sub-features of “GUI”: “TouchScreen” and “Internet”.

For Pattern E, we can imagine that the inhabitant receives a security alarm notification in one of the in-house touch screens. It may happen that the inhabitant does not want to spend time filling reports and receiving the security team to inspect the house. In addition, the inhabitant also knows that after X number of times by Y period of time, that s/he or the Smart Home calls the security company, s/he will have to start paying extra-money to the security company. Under these circumstances the inhabitant aborts the security notifications to the security company throughout the Touch Screen. Therefore, the goal of the “security” feature is not achieved because the process to secure the house was not completed and the inhabitants may be in danger in case of a real security alarm.

To avoid this feature interaction scenario the developer could decide that the security notifications will be sent first to the security company via Internet and then to the inhabitant. This implies to assure that the “secure the house” behavior is

completed before the inhabitant or anyone else tries to inhibit it. This is enforced to avoid that the inhabitant risks his security to save some money.

3.2 Applying the Overlapping Detection Strategy

The patterns in the previous section help to concentrate the detection of feature interactions on some features, based on coarse-grained relationships between use cases. These use cases give a hint where we should start designing use case scenarios. To find points of interaction between features in the requirements models, we look for fine-grained elements that are related to more than one feature and that may appear together in the same scenario. Figures 7-9 show the activity models for the use cases "Secure the House", "Open and Close windows", and "Smart Heating", which are related with the features "Security", "ElectronicWindows" and "SmartHeating", respectively. Also, part of the relationships, or trace links, between features and requirements model elements are summarized in Table 1.

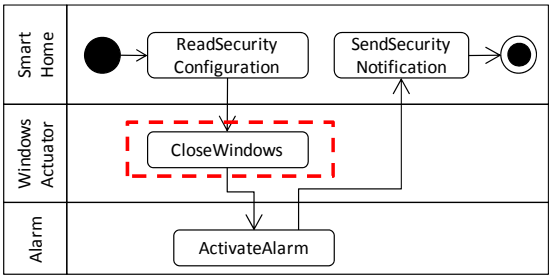


Figure 7. Secure the House Activity Model

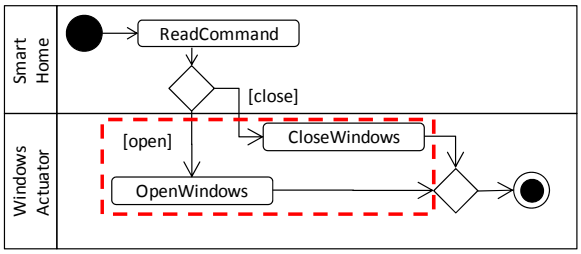


Figure 8. Open and Close Windows Activity Model

As it is shown in the activity models, and highlighted in the trace links table, "Windows Actuator" is shared by the features "Security", "Smart Heating" and "Open and Close Windows". So, according to this strategy, this is a symptom of a candidate feature interaction if the developer determines that there are actions that cannot be performed simultaneously in the identified shared element or resource.

This kind of feature interactions detected in the Smart Home system has been analyzed also by other authors. Khkpour et al. [13] called them *Action conflicts*. These conflicts require the identification of conflicting actions manually. For example lock and unlock window/door actions that are in conflict if they are triggered simultaneously. Nakamura et al. [14] refer to *Action Conflicts* as *Appliance Interactions*. Intuitively, an appliance interaction means that two methods have conflicting goals or post conditions that cannot be satisfied simultaneously on the common appliance [14]. A typical example are the methods "TV.on()" and "TV.off()". In our case study it would be "WindowsActuator.close()" and "WindowsActuator.open()" (marked with dashed lines in Figures 7-9). These methods are invoked in shared fragments of scenarios that describe the behavior of different features like Security and Smart Heating. Also *Appliance Interaction* can appear where the execution of one method may disable another. Khkpour et al. [13] also talked about *Inexecutable Action conflicts*. In the Smart Home this happens when the inhabitant aborts manually the security notification as described in Section 3.1.

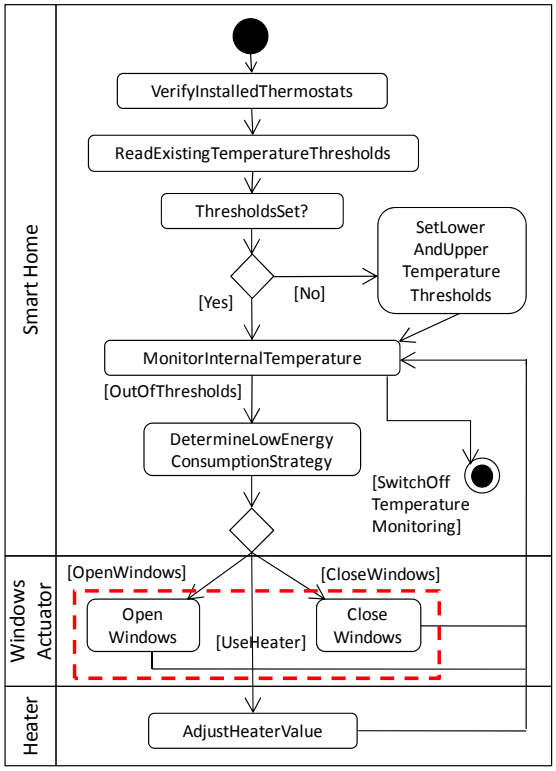


Figure 9. Smart Heating Activity Model

4. RELATED WORK AND DISCUSSION

Traditionally, feature interactions are associated with the telecommunications domain but recently have appeared in other domains like software systems. We believe that overlapping and dependency pattern detection may not be the best technique for the detection of feature interaction in arbitrary kinds of models in all domains. There are other approaches that deal with detection in specific domains and might offer more accurate results because of the detail of their models. For example, in SPLE there has been some work related with feature interaction detection. Classen [3] employs problem frames to analyze feature interactions between the environment and the SPL. This approach helps to reason about each interaction separately and requires a high degree on formalization of each interaction scenario. MATA uses the

Critical Pair Analysis (CPA) offered by its underlying composition tool AGG [15] to detect feature interactions. They focus on the detection of problems that avoid finishing the composition of the models of a product. For example, bad order of the compositions rules for the models that specify each feature.

To describe a precise technique to find feature interactions is not our goal in this work. This technique suggests the points or places in the models where we might find feature interactions. Since we cannot guarantee that the points that we discover are genuine FIs, we call them "candidate" points of FIs. Also, the patterns identified and described in this work naturally do not cover all the possible situations that indicate potential feature interactions. There are exceptional situations or "side effects" that might happen during the system execution that may be out of the scope of the two strategies mentioned applied in such high level descriptions of the functionality of the system used in requirements engineering. We focus on requirements models which are special since they do not contain many details about the structural components and the interactions between the high abstraction level modules of the system. Therefore, use cases and activity models are just the means that help us to analyze the functionality of a complex system looking at it from a high level end-user view.

We decide for a lightweight approach to find the points in which two or more features might interact. We consider the semantics of the relationships between the recurring requirements models like use cases, and overlapping detection between activity models that specify the behavior of the system. In particular, we believe that overlapping detection as a way to detect potential feature interactions may be achievable in SPLs because of the fact that many feature-related model fragments have a certain structural similarity with other model fragments [8].

The power of our approach is its simplicity, because it relies on the analysis of the relationships between elements in conventional SPL requirements models. This, in comparison with other approaches for feature interaction detection, does not require extra models formalizing each part of the behavior of the features. Overlapping and dependency patterns detection are useful to indicate where to find candidate points of feature interactions in the context software product lines. However, we think that the approach envisioned in this paper should be taken as a complement for the use of more formal and detailed detection approaches, not like the unique or optimal solution.

5. CONCLUSIONS AND FUTURE WORK

To identify candidate points for features interactions we focused on the identification of some patterns of dependency between features, and the overlapping between the models that design the features of the SPL. We are conscious that the overlapping detection strategy as the mean to identify points where there are potential feature interactions may not be the most accurate solution. Working with models requires discipline to keep updated trace links between each model fragment with the features that they are related to, and also to use standard ways to name the model fragments to make them comparable with other fragments. Our approach suggests that there is a trade-off between accurateness (formal and detail models to analyze feature interactions in requirements) and simplicity (dependency patterns and overlapping detection). Of course, we cannot judge for one or the other, nor conclude about a mixture of both yet. However, we believe that our work can help to initiate a discussion about this issue for feature interaction detection in model-oriented requirements approaches for SPL.

We need further investigation on the validation of our approach with more case studies. Also, we are working on the combination of more detailed techniques to specify the behavior of features and the lightweight strategies presented in this paper. Finally, we are working on conflicts that are derived from non-functional properties of the SPL, and their relationship with functional FIs.

ACKNOWLEDGMENTS

This work is supported by the European FP7 STREP project AMPLE [9].

6. REFERENCES

[1] Metzger, A. Feature Interactions in Embedded Control Systems. *Comput. Netw.*, 45 (5): 625-644, 2004.

[2] Combes, P. and Pickin, S. Formalisation of a User View of Network and Services for Feature Interaction Detection. In *Feature Interactions in Telecomunications Systems*, pages 120-135, Amsterdam, The Netherlands, 1994. IOS Press.

[3] Classen, A. *Problem Oriented Modelling and Verification of Software Product Lines*. Masters Thesis, University of Namur (FUNDP), Namur, Belgium, 2007.

[4] Nhlabatsi, A., Laney, R. and Nuseibeh, B. Feature Interaction as a Context Sharing Problem. In *10 Int. Conf. on Feature Interactions*, Lisbon, Portugal, 2009. IOS Press.

[5] Jayaraman, P., Whittle, J., Elkhodary, A. and Gomaa, H. Model Composition in Product Lines and Feature Interaction Detection Using Critical Pair Analysis. In *10 Int. Conf. on Model-Driven Languages and Systems*, volume 4735, pages 151-165, Nashville, USA, 2007. Springer.

[6] Xuan, H. and Xu, J. Web Services Feature Interaction Detection Based on Graph Transformation - A New Interaction Detection Method. In *Feature Interactions in Software and Comunication Systems X*, Lisbon, Portugal, 2009. IOS Press.

[7] Mehner, K., Monga, M. and Taentzer, G. Analysis of Aspect-Oriented Model Weaving. *Transactions on Aspect-Oriented Software Development V*, 5490: 235-263, 2009.

[8] Apel, S., Janda, F., Trujillo, S. and Kästner, C. Model Superimposition in Software Product Lines. In *Int. Conf. on Model Transformation*, Zurich, Switzerland, 2009.

[9] AMPLE. Ample Project, 2009. http://www.ample-project.net.

[10] Morganho, H., et al. Requirement Specifications for Industrial Case Studies. AMPLE Project, D5.2, 2008.

[11] Siemens AG - Research & Development, 2009. http://w1.siemens.com/innovation/en/index.php.

[12] Alférez, M., Santos, J., Moreira, A., Garcia, A., Kulesza, U., Araújo, J. and Amaral, V. Multi-View Composition Language for Software Product Line Requirements. In *2nd Int. Conf. on Software Language Engineering*, Denver, USA, 2009.

[13] Khakpour, N., Sirjani, M. and Jalili, S. Formal Analysis of Smart Home Policies Using Compositional Verification. In *Feature Interactions in Software and Communication Systems X*, Lisbon, Portugal, 2009.

[14] Nakamura, M., Igaki, H., Yoshimura, Y. and Ikegami, K. Considering Online Feature Interaction Detection and Resolution for Integrated Services in Home Network Systems. In *Feature Interactions in Software and Communication Systems X*, Lisbon, Portugal, 2009.

[15] Taentzer, G. AGG: A Graph Transformation Environment for Modeling and Validation of Software. In *AGTIVE*, Virginia, USA, 2003.

Author Index

Namik Aktekin, 103
Mauricio Alferez, 117
Vasco Amaral, 117
Sven Apel, 27, 55, 63
Joao Araujo, 117
Mohsen Asadi, 95

Jorge Barreiros, 43
Don Batory, 1
Matthias Blume, 3
Marko Boskovic, 95
Götz Botterweck, 109

Wonseok Chae, 3

Josune De Sosa, 87

Christoph Elsner, 35

Janet Feigenspan, 55

Dragan Gasevic, 95
Xiaocheng Ge, 49
Sebastian Günther, 11

Marek Hatala, 95
Florian Heidenreich, 69
Peter Höfner, 75

Bo Nørregaard Jørgensen, 19

Christian Kästner, 27, 55
Nima Kaviani, 95
Stefan Kowalewski, 109
Martin Kuhlemann, 27
Uirá Kulesza, 117

Thomas Leich, 27, 55, 63
Christian Lengauer, 63
Jörg Liebig, 27, 63
Daniel Lohmann, 35, 81

Ricardo Mateus, 117
John McDermid, 49
Xabier Mendialdua, 87
Bardia Mohabbati, 95
Ana Moreira, 43, 117
Bernhard Möller, 75

Andrzej Olszak, 19

Richard Paige, 49
Andreas Pleuss, 109
Andreas Polzer, 109

Wolfgang Schröder-Preikschat, 35, 81
Julio Sincero, 81
Sagar Sunkle, 11

Reinhard Tartler, 81
Bedir Tekinerdogan, 103
Salvador Trujillo, 87

Ander Zubizarreta, 87