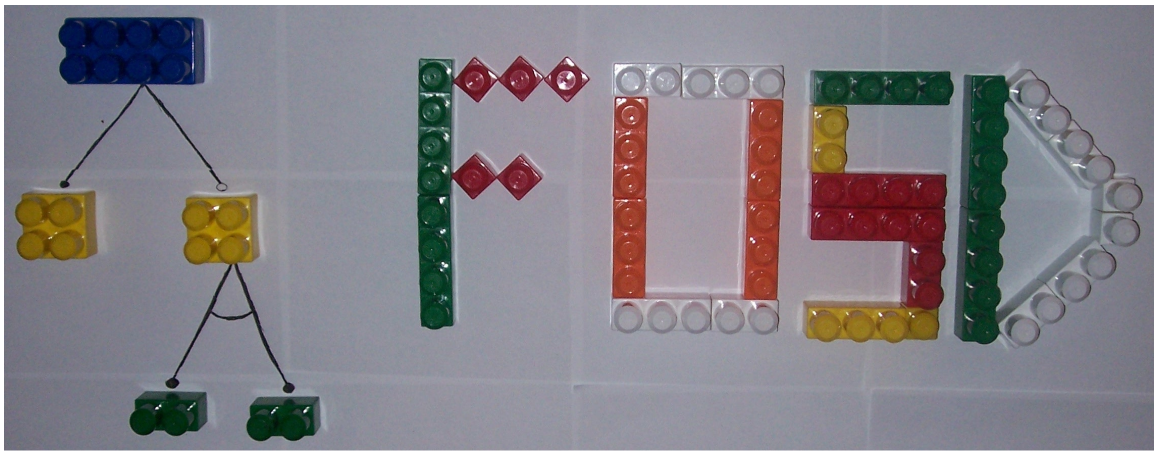


Proceedings of the 2nd International Workshop on Feature-Oriented Software Development (FOSD'10)

October 10, 2010 – Eindhoven, The Netherlands

Editors:
Sven Apel *(University of Passau, DE)*
Don Batory *(University of Texas at Austin, US)*
Krzysztof Czarnecki *(University of Waterloo, CA)*
Florian Heidenreich *(University of Dresden, DE)*
Christian Kästner *(University of Marburg, DE)*
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: 978-1-4503-0208-1

Preface

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

The goal of the FOSD'10 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 Christian Prehofer, 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 previous edition, held at GPCE'09, and a series of workshops on product lines, generative programming, and aspect orientation, held at GPCE'06, GPCE'07, and GPCE'08.

We received an excellent number of 20 submissions. The review process led to the selection of 11 papers for presentation. We would like to thank the program committee and the authors for their contributions to the success of the workshop.

Program Committee

Table of Contents

Feature Oriented-Programming: Back to the Future

Christian Prehofer
Fraunhofer ESK
Munich, Germany
christian.prehofer@esk.fraunhofer.de

ABSTRACT

Feature-oriented programming has its origin in the old quest of computer science: modular composition of software "artifacts" in software development. While there exist many notions of modularity and composition, feature-oriented programming tackles the case of highly-entangled software features where typical module or component concepts fail. The reasons for this are feature interactions and dependencies which make it very hard to write modular software which can be composed in a flexible way. In contrast to other efforts like aspect-oriented programming, feature-oriented programming has from the beginning focused on the semantics or behavior of software, not just on code modularity.

In this presentation, we review the origins of feature composition and feature interactions, which first occurred as an explicit research problem in telecommunication software. We argue that feature interactions are pervasive in many areas of software development and are a common source of modularity and quality problems. Then, we present more precise formalization of modularity and compositionality of features. We discuss when adding features is modular or "harmless" from a semantic or behavioral point of view. In particular, we focus on modularity and interaction for multiple feature compositions. A future challenge for feature-oriented software is the graphical modeling of features and feature interactions. As we aim to have modular composition for graphical models and to generate code from these graphical artifacts, it is important to have precise semantics and clear refinement concepts. For this, we discuss how features and interactions can be represented by state-based specifications in a graphical way. Refinement concepts are presented which allow the flexible composition of features and interactions in statechart diagrams.

Categories and Subject Descriptors

D.3.3 [**Programming Languages**]: Language Constructs and Features

General Terms

Languages, Theory

Keywords

Feature-oriented programming, modularity, composition, feature interactions, keynote

Automating Energy Optimization with Features

Norbert Siegmund
University of Magdeburg
Magdeburg, Germany
nsiegmun@ovgu.de

Marko Rosenmüller
University of Magdeburg
Magdeburg, Germany
rosenmue@ovgu.de

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

ABSTRACT

Mobile devices such as cell phones and notebooks rely on battery power supply. For these systems, optimizing the power consumption is important to increase the system's lifetime. However, this is hard to achieve because energy-saving functions often depend on the hardware, and operating systems. The diversity of hardware components and operating systems makes the implementation time consuming and difficult. We propose an approach to automate energy optimization of programs by implementing energy-saving functionality as modular, separate implementation units (e.g., feature modules or aspects). These units are bundled as *energy features* into an energy-optimization feature library. Based on aspect-oriented and feature-oriented programming, we discuss different techniques to compose the source code of a client program and the implementation units of the energy features.

Categories and Subject Descriptors

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

General Terms

Algorithms, Design

Keywords

Software Product lines, Energy consumption, Feature-oriented programming

1. INTRODUCTION

The widespread use of battery-supplied systems such as notebooks and mobile phones leads to a design shift toward energy optimization. Increasing the lifetime of a system is considered more important than an optimal performance. Moreover, due to increasing energy costs, energy optimization techniques such as virtualization are also in the focus of research for systems with direct power supply [37]. It is expected that energy cost of servers will soon exceed the purchase cost of the servers [7, 33]. Beside improvements in hardware architecture, the efficient use of hardware by software is a key factor to reduce energy consumption.

Consequently, various compilers have been developed to reduce energy consumption (e.g., by instruction reordering and loop transformation [38, 16, 13]). Furthermore, operating systems provide their proprietary functionality to optimize energy consumption. For example, Windows Mobile[1] comes with an integrated power-management-component that can be used by applications to set different power modes for hardware components within the device. The Advanced Configuration and Power Interface (ACPI) defines an open standard for the power management related configuration of a system's hardware [19]. It enables applications to tune the energy consumption according the current workload. Laptops, desktops, and other hardware systems are deployed with this interface. However, in its current specification,[2] it consists of 727 pages, which reflects the complexity developers have to face and the required implementation effort for its realization.

Although there is a considerable number of energy optimization strategies, it has been observed that many applications do not use them [18, 25]. Reasons are heterogeneity of hardware with its unique energy-management functionality as well as complexity of software solutions (e.g., for ACPI) [4]. Adapting a software to different hardware devices and operating systems often increases development time and cost. Business constraints and release deadlines hamper the implementation of energy optimization techniques further.

The problem of handling variability and heterogeneity of hardware and operating systems in software development is not new. *Software product line (SPL)* engineering has shown to be a suitable approach to tackle this problem [12, 34]. An SPL consists of a set of features that represent end-user visible characteristics of a software. A user generates a tailor-made programs by selecting features that satisfy her requirements. This way, different programs can be generated based on a common code base.

We propose an approach to automate the application of energy-saving techniques (referred to as *energy features*) based on SPL engineering. The energy features are bundled in a *feature library*. The library consists of energy optimization functions for different hardware and operating systems. For example, a feature implements the functionality to activate and deactivate the WLAN for the Windows Mobile OS. By selecting energy features according to given system and user requirements, we can apply energy-saving functions to a target program. We use the feature library to achieve an improved separation of concerns regarding energy-saving techniques. Furthermore, the library allows us to reuse the energy features in different programs and to cope with the required variability (e.g., different hardware and operating systems). We make two contributions: (i) We provide means to ease the development of

[1]We consider OS version 5 and 6.
[2]Revision 4.0: `http://www.acpi.info/`

energy-efficient software and (ii) we propose a method for creating a library of energy-saving techniques. The expected advantages are:

- With the feature library, developers do not require deep knowledge about energy optimization. Energy-efficient features can be added without investigating which energy-saving technique is available for a particular system.
- Reusing energy features of the library reduces the development time of new products.
- We provide a simple configuration interface to utilize energy-reduction techniques. This allows even non-expert developers to optimize their programs.

We think that the energy feature library is needed to handle the diversity of techniques and make them available for practical use. As we describe see in Section 2.3, the kinds of optimization technique can heavily differ. Due to these differences, a variety of interfaces are required to make the features usable. In our current work, we concentrate only on those energy-saving techniques that are accessible by means of method calls. These techniques are commonly cohesive and generally applicable so that a reuse for different programs is possible. We show how feature-oriented programming [5] and aspect-oriented programming [23] can be used to compose a program with energy features. Furthermore, we present an approach to use energy-saving techniques by means of code instrumentation to access energy-management code.

This paper discusses ideas to apply feature-oriented software development to new domains such as energy optimizations. Our work concentrates on challenges that arise during design time of a software. Although, this work is at a theoretical level, it identifies problems that exist in other domains and present possible solutions.

2. BACKGROUND

Our approach is based on a library of features that implement different energy optimization techniques. We use *feature-oriented programming (FOP)* [35, 5] as a technique for modularizing the code that implements a feature. Next, we give a short overview of modeling features and how they can be implemented using FOP. Finally, we present some important representatives of energy-saving techniques.

2.1 Feature Modeling

We use a *feature model* to design the features of our library and their relations. A feature model is a hierarchical structure consisting of features and their constraints among each other [22, 14]. Figure 1 shows a feature model for the energy feature library. Mandatory features (e.g., feature *OS* of Figure 1) have to be included in every variant (denoted with a filled bullet) when the parent feature is selected. Optional features (denoted with an empty bullet; e.g., feature *Backlight*) are not required for each variant; it is up to the user to select them during the product configuration process. There are also groups of features representing additional variability. *Alternative groups* require the selection of exactly one feature of the group (not more than one); *or groups* allow the selection of one or more features of the group.

2.2 Feature-Oriented Programming

FOP is a technique that enables the implementation of features as fundamental modules of abstraction and composition [5]. This means, a feature exactly corresponds to a *feature module*. A feature module encapsulates the source code otherwise scattered across different classes inside a single modular code unit. It can define new classes or extend existing classes (as *refinements*) originally defined by other feature modules. To derive a program, a set of feature modules is incrementally composed. This composition obtains final classes from classes and their refinements. In Figure 2, we show the architecture of a database management system (DBMS) implemented with FOP. Horizontal layers represent the features of the DBMS and the vertical bars represent classes. Starting from a base implementation of a class, multiple refinements, which belong to different features, are added resulting in a *layered design*. Refinements add new members to classes such as methods and attributes or extend existing methods with additional functionality. For example, if a user selects feature *Query Engine*, class *DB* would consist of the base implementation and a refinement to handle SQL queries.

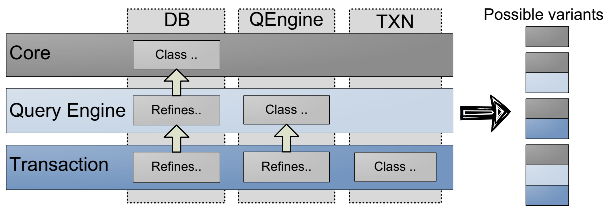


Figure 2: Decomposition of classes (vertical bars) with respect to features (horizontal layers).

2.3 Energy-Saving Techniques

There are different kinds of energy-saving techniques. We divide them into two classes: code transformation techniques and hardware- and operating-system–dependent techniques. Before we describe the two classes, we clarify the differences between power and energy, because it is necessary for understanding the techniques. Unfortunately, power and energy are often used interchangeably in literature [40]. The consumed power P of a system is the consumed energy E per time unit t:

$$P = \frac{E}{t} \tag{1}$$

Given that a system consumes energy[3], the intended goal is to save energy for a given task (in contrast to saving power). Considering Equation 1, reducing the power consumption may also reduce the energy consumption, but not necessarily. For instance, a task can either be performed with high power in a short time period or with low power, but for a longer time period. It depends on the task and algorithm what saves more energy. For example, constant time tasks should be performed with low power.

Code Transformation Techniques. The first category of energy optimization techniques contains different code transformation approaches. We first describe some solutions based on compilers and continue with source-to-source transformations that are compiler independent. Compiler approaches transform a software such that the energy consumption is reduced. One of the first solutions was proposed by Tiwary et al. [38]. The main idea is to reorder processor instructions such that less switches in CPU registers are necessary. Furthermore, they show how a reduction of memory operands in a program can lead to energy savings. Other optimizations are based on method inlining and loop unrolling [26] as well as on an energy efficient scheduling of a program's instruction set [39]. A more recent approach allows developers to define

[3]For example, batteries are charged with a certain amount of energy instead of power.

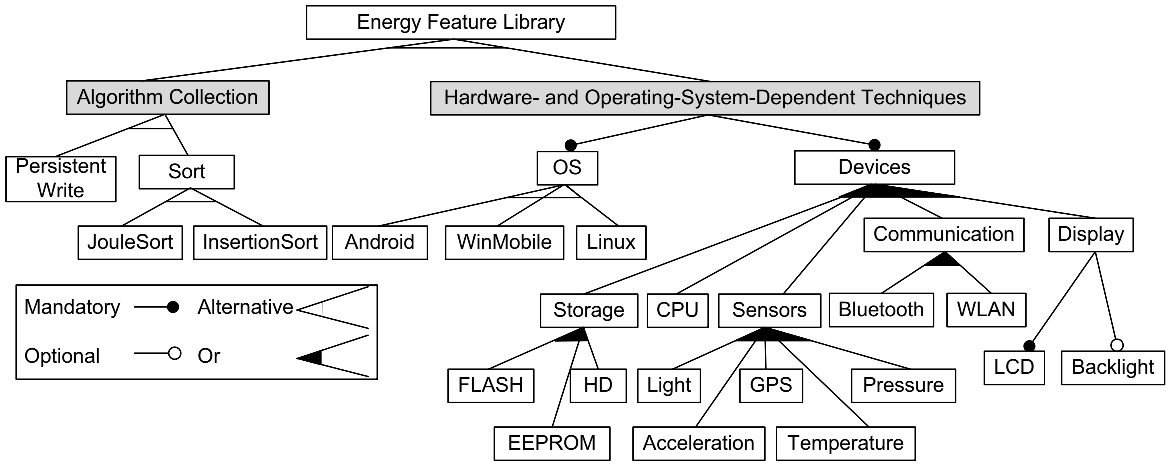


Figure 1: Feature model of the energy feature library.

objective functions for the compilation process [13]. Based on the objective function, the compiler selects suitable code transformations out of a pool.

Beside compiler-based optimizations, source-to-source transformations can also save energy. Fei et al. have shown that transforming the source code based on the analysis of the program's control flow can achieve energy savings of up to 37.9% [15]. The transformations reduce inter-process communication (reduces OS overhead for context switches), minimize the number of concurrent processes, and reallocation of computations of one process to another (reduces synchronization between processes).

Another study provides insights into energy consumption of language constructs in the C# programming language [10]. For example, dynamic anonymous methods consume 3 to 4 times more energy than dynamic methods. Another interesting fact is that protected attributes require two times more energy than private or public attributes. Of course, this may be different in other runtime environments; however, we observe that simple changes in the source code can have a large impact on energy consumption.

Beside application-independent transformations of the source code, the selection of a suitable algorithm is often more important for large energy savings. Researchers proposed energy-efficient algorithms such as for sorting large amounts of data [36] as well as studies about choosing the optimal algorithm to reduce energy consumption [11, 9] which can even be taken a step further to combine hardware-dependent techniques with energy-efficient algorithms. For example, Pisharath et al. reduces energy consumption of queries in an in-memory database system by selectively setting different memory banks into a certain power state [31]. Controlled by an intelligent query engine of the database system, energy consumption could be decreased by up to 68%. Solutions that are very specific for certain application scenarios are very difficult to encapsulate into a single implementation unit. Thus, reuse in different programs is unlikely. Therefore, we currently focus only on generic algorithms for the feature library that fulfill reoccuring tasks in different programs.

Hardware- and Operating-System–Dependent Techniques. Beside code transformation techniques, software can also efficiently control hardware components to save energy. The most prominent techniques are dynamic voltage scaling [29, 21] and frequency scaling of CPUs [30, 32]. Both approaches are used to reduce the power when a system has a low workload. Since the power *P* depends proportionally on the frequency *f* and the voltage *v* squared, reducing the frequency as well as the voltage decreases the used power:

$$P \propto f * v^2 \tag{2}$$

For constant-time tasks, reducing power consumption proportionally reduces energy consumption (as it follows from Equation 1). Therefore, both factors (cf. Equation 2), frequency and voltage, should be adjusted depending on the current workload of the system. Some approaches analyze applications statically to include the scaling commands into the application (e.g., when certain events occur) [41]. Other solutions dynamically change the frequency or voltage in a pre-defined time interval [3]. These techniques use profiling mechanisms to predict the future workload of the system so that dynamic adaptations are possible. Currently, we do not address this kind of optimization because it is difficult to provide a general applicable approach that saves more energy than already existing solutions provided by operating systems.

ACPI is an industry-driven specification for power management of hardware devices. It allows the operating system to control different power states of the hardware. For example, a device could be set in four different states; beginning with a full powered on state (*D0*) and ending with powered off state (*D3*). The different states can be set by programs by means of operating system API calls.

Operating systems for mobile devices provide special power management solutions similar to ACPI. For example, Windows Mobile comes with its own power management.[4] Applications can control the power states of different devices such as the backlight, the screen, etc. Setting the device into a specific state requires API calls to the operating system. Another example is the *Android*[5] power management that requires *Wake Locks* to request CPU resources. If there is no wake lock active, the CPU will shut down (and saves energy). As Windows Mobile, Android supports different power states. Unfortunately, setting the device into a specific state differs in both operating systems.

The described operating-system techniques are used by means of

[4]For detailed information, see `http://msdn.microsoft.com/en-us/library/aa923906.aspx`.
[5]`http://www.android.com/`

API calls. Thus, a feature that utilizes the provided energy management functionality of operating systems can be easily implemented as a separate, self-contained implementation unit. The energy feature library contains especially such features because a reuse between different programs is easy to achieve.

3. TOWARDS AN ENERGY FEATURE LIBRARY

Due to the multitude of possible energy-saving techniques, it is hard for developers to know every possible mechanism wasting a large quantity of possible energy savings. This problem even grows if the target hardware and operating systems varies for the same program. In this section, we present our approach to cope with the heterogeneity of techniques by developing a library of energy-saving technique. Our idea for an automated optimization is the integration of code of the feature library in a client program. Depending on the execution of a feature's code of the client program, we (i) replace this code with the code of the library or (ii) (de)activate hardware components. Unfortunately, we cannot implement every mechanism to reduce energy consumption with this approach. For example, compiler-based optimizations cannot be applied with a feature library.

Thus, we include only operating-system-dependent techniques as well as some code transformations in the energy feature library. However, the idea is that the library can be extended by others providing their own techniques. Next, we describe the envisioned features of the library. We illustrate how application engineers can use the energy feature library to select proper energy-saving techniques for their program.

3.1 Modeling the Energy Feature Library

We use a feature model to relate the different techniques for energy management. This allows us to define a view on the energy optimization domain as well as to use the feature model for the configuration process. In Figure 1, we depict an initial proposal for the energy feature library. The energy feature library contains of two main parts: *Algorithm Collection* and *Hardware- and Operating-System–Dependent Techniques*.

Algorithm Collection. We create a library of algorithms that minimize the energy consumption. Subfeatures of *Algorithm Collection* implement common, recurring algorithms in an energy-efficient manner. For example, feature Sort in Figure 1 contains two subfeatures that sort data with a minimal energy consumption [36, 8]. Developers of different domains can further add their domain-specific algorithms to enrich this algorithm library. Possible examples are energy-efficient read and write of data from and to persistent storage [20], different data aggregation strategies for network communication [24, 6], or data caching algorithms to reduce communication [28]. Although a large amount of such algorithms are often only reusable in a certain domain, we expect that the benefit for the domain is still considerable.

The idea of algorithm collection is to replace the algorithms in the target application with the algorithm features of the *Algorithm Collection*. This way, we can exchange an algorithm of a program with another one that has an equal functionality but requires less energy. To use an algorithm, a developer has to encapsulate the corresponding algorithms with a proper interface in her program. For example, if a developer wants to exchange the algorithm for sorting data, she has to separate the sort algorithm into a single implementation unit, i.e., a feature module. This implementation unit can then be replaced by the feature module of the algorithm collection (e.g., feature *InsertionSort*). If the interface of the algorithms do not match, developers have to implement an adapter to integrate the algorithm of the library into the program. The rational behind different sorting algorithms is that the efficiency of a sorting algorithm regarding energy consumption depends on the amount and kind of data to be sort. Depending on these factors, a user selects the sorting algorithm that fits best to her workload.

Hardware- and Operating-System–Dependent Techniques. To ease the selection of energy-saving techniques, we use features for operating systems (feature *OS* and subfeatures) and for hardware components (feature *Devices* and subfeatures). A selection of these features together map to the feature module(s) that implement the energy-saving technique(s). In detail, feature *OS* provides energy management functions for different hardware components as described in Section 2.3. Since these energy management functions strongly dependent on the used device, we have to model the hardware, too. The possible available hardware components of a system are represented as subfeatures of feature *Devices*, in Figure 1. Together with the selection of the operating system, we can select a feature module that efficiently utilizes the hardware. For example, if a developer selects feature *WinMobile* and feature *Backlight*, we include the feature module in the program that powers the backlight on and off. The place for inclusion is defined separately, as describe below. In Section 4, we show the usage of features *WinMobile* and *WLAN*. Selecting the two features enables a program to trigger the power mode of a WLAN device depending on the current executed functionality. Overall, selecting the subfeature of feature *OS* defines which operating system API calls have to be composed and selecting a subfeature of feature *Devices* defines which hardware component has to be controlled by the program.

3.2 Mapping Energy Features to Application Functionality

To use energy features, we need a mapping between the functionality that has to be optimized and the feature module of the energy feature. Thus, a developer configures the energy feature library for client application feature that has to be optimized. This means, each feature of the client program maps to a configuration of the library. Based on this configuration, we derive the correct feature module that maps to the functionality of the program. In Figure 3, we show such a mapping between a program and the energy feature library. In this example, the program is a variant of an SPL for data management solutions targeting resource-constrained systems. On the left side of Figure 3, we depict the features of the DBMS SPL that should be optimized regarding energy consumption. For each feature, we configure the energy feature library (right side of Figure 3) using the feature model of Figure 1. The configuration of the energy library results in a set of feature modules that realize the energy-saving functions. These feature modules are mapped to the corresponding modules of the DBMS SPL (center of Figure 3). For example, to power the WLAN device only when feature *Distribution* of the DBMS SPL is executed, we create a mapping between this feature and the feature module *W_W* which is derived from the configuration of the energy feature library. The result is that the energy management for the WLAN device is controlled by the program. The device is powered only when the program uses its communication functionality. Establishing such mappings is a straightforward process. It requires only little expert knowledge about energy management of operating systems and hardware components.

If a developer wants to use energy-efficient algorithms, she has to create a mapping between the algorithm feature of the algorithm

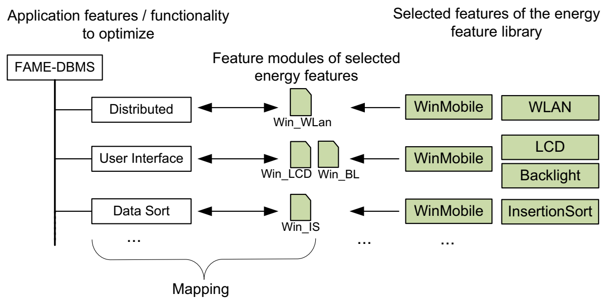

Application features / functionality to optimize
Selected features of the energy feature library
FAME-DBMS
Feature modules of selected energy features
Distributed
Win_WLan
WinMobile
WLAN
User Interface
Win_LCD
Win_BL
WinMobile
LCD
Backlight
Data Sort
Win_IS
WinMobile
InsertionSort
...
Mapping

Figure 3: Mapping application features to energy features.

Feature Module WLAN_WinMobile_On (Win_WLan)

```
1  refines class [CLASS_NAME] {
2    [RETURN_TYPE METHOD_NAME]([ARGS]) {
3    //Turns the WLAN device on
4    _energySplClass.SetDevicePower
5      (_energySplClass.getWirelessDeviceName(),1,0);
6
7    //Execute the send functionality
8    [RETURN_TYPE RETURN_VALUE=]super::[METHOD_NAME(ARGS)];
9
10   //Turns the WLAN device off
11   _energySplClass.SetDevicePower
12     (_energySplClass.getWirelessDeviceName(),1,4);
13   [return RETURN_VALUE];
14  }
15 };
```

Figure 4: Skeleton of energy-optimization feature modules (FeatureC++ example).

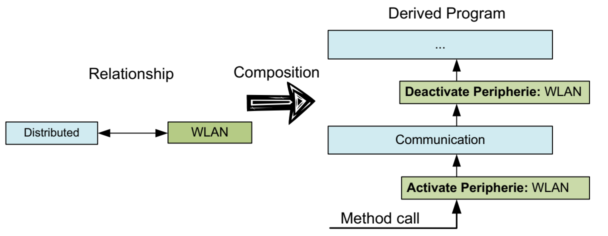


Figure 5: Composing application features with energy optimization code.

collection and the algorithm feature of her program. For example, the mapping between feature *Data Sort* and feature module *W_IS* in Figure 3 results in a DBMS variant that uses the InsertionSort algorithm for sorting data; instead of the original implementation. Beside the reduced energy consumption, the developer has to take additional effects on non-functional properties into account. For example, by applying feature *InsertionSort* the performance may be decreased. If performance is more important than energy consumption, the mapping should not be established.

4. PRODUCT GENERATION

A key contribution of our approach is the application of energy-saving techniques to programs with a very small implementation effort for the developer. The technique requires only a mapping between features of the application SPL and features of the energy feature library. Depending on the selection of an algorithm feature or an hardware-dependent feature, different composition mechanisms have to be used. For algorithm features, we have to exchange the implementation unit of the program with a feature module of the feature library. When using FOP, this is a straightforward process because either the existing feature modules must be physically replaced or the configuration of the application SPL must be changed accordingly. For hardware-dependent techniques, additional code from the library must be integrated into the program. This requires new implementation techniques to connect application features with energy-optimization code. We propose three different implementation variants: *Manual Layered Composition*, *Automated Layered Composition*, and *Energy Manager*. We explain all approaches in detail and discuss their advantages and drawbacks.

4.1 Manual Layered Composition

The first approach to realize the integration of energy optimization code is using the layered composition process of FOP. Features modules are represented as layers. Similar to classes, methods are composed from multiple refinements resulting in a refinement chain executing the functionality of all existing refinements sequentially. This way, functionality of the method from different feature modules is executed depending on the order of the refinement chain (or layers). To access the energy management, a developer has to implement a feature module that refines the application's functionality. This means, a developer has to write method refinements (consisting glue code) for those methods that have to be optimized regarding energy consumption. To simplify manual implementation, we envision a skeleton that enables a semi-automated generation process. Such a skeleton can provide the implementation of the energy code for each feature of the energy feature library. It requires only the declaration of the method that is the target for energy optimization. In Figure 4, we show an example for such a skeleton based on FeatureC++[6]. FeatureC++ is a C++ language extension for FOP [1]. It comes with a source-to-source compiler. FeatureC++ uses keyword **super** (Line 9) to execute the next (above) method refinement of the composed method. In square brackets, a developer has to include the name of the class and method (Lines 1-2) that has to be refined with the energy-saving code. Furthermore, if there is a return value, it has to be stored temporarily (Line 8) and returned after the device is powered off (Line 13). The benefits of this approach are the fine granularity of changes to the program's code (i.e., only selected methods are surrounded with energy management code) and the direct control of code changes by developers. This allows developers to have a direct influence on the code changes.

4.2 Automated Layered Composition

Another solution to compose the energy features with the program is based on aspectual feature modules[7] [2], as illustrated in Figure 6. The idea is to enclose the code of the program's feature with code of the feature library (cf. Figure 5) to manage hardware components. In this solutions, features are implemented as aspectual feature modules. An aspectual feature module can introduce aspects, method refinements, and new classes to a program. The aspects are weaved into the application to access the energy management code. The energy management code can be implemented, in turn, as classes, also part of the aspectual feature module. In order to access the energy management functionality, we use an around advice (Line 2–3) for each method belonging to a

[6]http://fosd.de/fcc

[7]Aspectual feature modules combine the concepts of FOP and aspect-oriented programming (AOP) [23].

Aspectual Feature Module WLAN_WinMobile (Win_WLan)

```
1  aspect WLAN_Energy_Management {
2   pointcut exWLAN() = execution("%.Send_Receive%(...)");
3   advice exWLAN() : around (){
4    //turn wlan on
5    _energySplClass.SetDevicePower
6    (_energySplClass.getWirelessDeviceName(),1,0);
7
8    proceed();//Execute the send functionality
9
10   //turn wlan off
11   _energySplClass.SetDevicePower
12   (_energySplClass.getWirelessDeviceName(),1,4);}
13 }
14 class EnergySplClass {
15  //Load ossvc library
16  EnergySplClass(){..}
17  //Set different power modes
18  SetDevicePower(..){..}}
19 }
```

Figure 6: An Aspectual feature module to apply WLAN (de) actionvation for feature Send_Receive (FeatureC++ example).

feature (of the application SPL) in a mapping. Depending on the energy-optimization technique, the piece of advice activates or deactivates different hardware. For example, we activate the WLAN device in Lines 5–6 and deactivate it in Lines 11–12 (cf. Figure 6). The whole aspectual feature module is stored in the library. The only part which needs to be generated is the pointcut for each public method of the program's feature. In Figure 6, we include the *Send_Receive* feature name in the execution pointcut. In order to select join points of aspects of a single feature, the feature has to be part of the pointcut definition.

4.3 Energy Manager

Another implementation technique is based on an energy manager that stores the mappings between features of the application SPL and the library. The energy manager is statically composed with the program including the mappings. It is accessed each time when the program execution reaches the code of a feature. Before feature code is executed, the manager checks whether it has to activate or deactivate a certain device. For example, when the DBMS calls the *send* method of feature *Send_Receive*, the manager is called to check for existing mappings. In this case, we have to execute the energy management to turn the WLAN on. To query the manager for mappings, we have to integrate a method call at the entry points of every method refinement of a feature. Such a code instrumentation techniques is also used in dynamic AOP approaches in which hook methods are generated to enable execution of additional code at runtime [17, 27]. FeatureC++ can be used to generate such method calls during the composition process for each program feature, which is the main difference to the aspectual-feature-module solution. The method calls were originally intended to support activation and deactivation of features at runtime. We use this technique to include a method *OnExecute(FEATURE_NAME)*. This method accesses the energy manager to evaluate if there is an existing mapping for the current feature. If present, this code is executed (e.g., the WLAN device is powered on). The device is turned off when the execution of the refinement has finished. This can be achieved by generating a method *OnExecuteExit(FEATURE_NAME)* before each return statement.

4.4 Discussion

Reuse of Energy Features. Usually, features are reused within different variants of a single software product line. In such a case, reuse is the main goal of the design and architecture of features. When using a library for energy optimization techniques, features have to be reused in very different programs. For common application features, a reuse would be very difficult to achieve. Obvious reasons are application-dependent interfaces, heterogeneous data structures, etc. However, the characteristics of energy optimization techniques suggest that their reuse in very different programs is possible. Energy features that rely on code transformation, are very generic. There is no application specific code in such a feature. Features for utilizing the energy management for different hardware components are often cohesive. That is, turning a hardware component on and off does not require complex adaptations of the application. An example for this case is ACPI that provides an interface for energy management accessible in different operating systems.

Rapidly Switching Between Different Features. A problem arises if we quickly enter and leave the code of a feature that belongs to a mapping. Putting a device in a certain power mode consumes time, the performance decreases, and energy is wasted. A possible solution is to use timers for the deactivation of hardware. The time interval for deactivation should be defined during the configuration and is stored together with the mapping. A finer granularity may be used to set different timers for individual features.

Performance. The execution of additional code due to the integration of the energy features may decreases the performance. This is especially the case when using the energy manager implementation because we execute the *OnExecute()* method for each feature. Hence, there is a trade-off between performance and energy optimization. We have to evaluate how large the impact of such a realization is. We will address this issue in further work. Nevertheless, when an application performs a constant time task, performance does not matter. This means, when we decrease the performance, the performed task may not require more time to finish. For these tasks, applying energy optimizations which come with small performance degradation will not have a negative effect.

Architectural Adaptations. We need a feature model of the program to create the mapping between program features and energy features. In some cases, a program's feature has to be decomposed into two features when it is not feasible to apply the energy management to the whole feature. For example, feature *Distribution* of the data management SPL consists of various functions that realize data distribution for different databases. Only a small part of this feature actually requires the WLAN device to send and receive data. Thus, only this part should be mapped to an energy feature of the library. Such an architectural change represents a way to apply the energy optimization code only at the point where it is really needed. Thus, there is a trade-off between a good program design and energy optimization. With additional implementation effort by the application developers the manual layered composition and automated layer composition can solve this problem. For example, pointcuts can be manually defined to address only the important methods. This hampers a complete automated generation, but may be more appropriate than a restructuring of the application SPL. The first approach of the layered implementation already requires

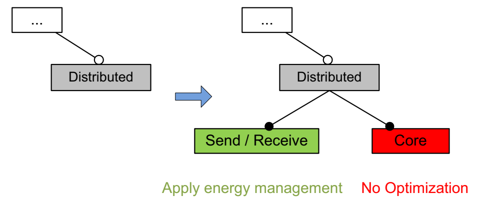


Figure 7: Dividing feature Distribution into two subfeatures to apply the optimization only to the Send / Receive functionality.

a partial manual implementation of energy feature modules.

Handling Method Calls Inside Encapsulated Methods. If the control flow leaves the functionality of a feature with energy management before it is completely executed, we may waste energy because hardware components only are powered off at the end of the feature's functionality. As an example consider the *Send_Receive* feature, if we have to wait for user input (e.g., to type in a password) in the send method, we might have to wait a long period of time and thus waste energy. The question is whether to turn the device off before leaving the method or keep it active. We think that an appropriate decomposition of application SPL features can reduce the occurrence of such problems.

Application SPL. Currently, we consider programs developed as SPLs, i.e., decomposed into features. However, this is not necessarily required. We only need a description of the functionality of the program that maps one-to-one to implementation artifacts (e.g., a feature model or a component model). We use this description to attach energy optimizations to the application's functions. This way, we can determine which functionality requires which device and can thus be optimized regarding energy consumption. To sum up, a serious reduction of energy consumption may require a design shift of software development towards energy optimizations. This means, a program needs an appropriate design of its features with respect to energy optimization (as we already discussed).

5. CONCLUSION

We presented an approach that allows programmers to use energy-saving techniques without the need of special knowledge about energy optimization. Developers do not have to invest time to learn different energy saving techniques and to apply them to their programs. We use software product line (SPL) techniques to model and implement different energy-saving techniques such as operating-system- and hardware–dependent functions. The functionality is implemented as separate implementation units (feature modules) and bundled in an energy feature library. Researchers and developers can add their own energy-efficient algorithms as features to the library.

To integrate the energy features into the program, we use mappings between program functionality and energy features. For example, a feature for sending data in a software maps to a WLAN feature of the energy feature library. Such a mapping is used to compose the source code of the energy features with the code of the program. We presented three approaches to implement this composition. The first two approaches use the layered design of feature-oriented programming. Layers represent features (or functionality) of a program. A program's feature can be composed with a feature of the library. The composition can be performed semi-automated (based on a skeleton class that contains energy-optimization code) or automatically by generated pointcuts of aspects. Furthermore, code instrumentation can also be used to insert an access method each time the program reaches feature code. The access method calls an energy manager that checks for existing mappings. If a mapping of the current feature exists the related energy optimization is executed.

In future work, we will evaluate the different approaches regarding energy-savings and the impact on performance. Furthermore, we will analyze if such an approach can also be used to optimize other non-functional properties such as performance and memory consumption.

Acknowledgement

Norbert Siegmund is funded by the ministry of education and science BMBF, number 01IM08003C. Marko Rosenmüller is funded by the German research foundation, project number SA 465/34-1. Apel's work is supported in part by the DFG projects #AP 206/2-1 and #AP 206/4-1. The presented work is part of the ViERforES[8], MultiPLe[9], FeatureFoundation[10], and SafeSPL projects.

6. REFERENCES

[1] 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, Sept. 2005.

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

[3] A. Azevedo, I. Issenin, R. Cornea, R. Gupta, N. Dutt, A. Veidenbaum, and A. Nicolau. Profile-based dynamic voltage scheduling using program checkpoints. In *Proceedings of the International Conference on Design, automation and test in Europe*, page 168. IEEE Computer Society, 2002.

[4] L. A. Barroso. The price of performance. *Queue*, 3(7):48–53, 2005.

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

[6] L. Becchetti, A. Marchetti-Spaccamela, A. Vitaletti, P. Korteweg, M. Skutella, and L. Stougie. Latency-constrained aggregation in sensor networks. *ACM Trans. Algorithms*, 6(1):1–20, 2009.

[7] C. Belady. In the data center, power and cooling costs more than the it equipment it supports. *Electronics Cooling*, 13(1), 2007.

[8] C. Bunse, H. Hopfner, E. Mansour, and S. Roychoudhury. Exploring the energy consumption of data sorting algorithms in embedded and mobile environments. In *Proceedings of the International Conference on Mobile Data Management: Systems, Services and Middleware*, pages 600–607. IEEE Computer Society, 2009.

[9] C. Bunse, H. Höpfner, S. Roychoudhury, and E. Mansour. Choosing the "best" sorting algorithm for optimal energy consumption. In *Proceedings of the International Conference*

[8]http://vierfores.de
[9]http://fosd.de/multiple
[10]http://fosd.de/ff

on Software and Data Technologies (ICSOFT), pages 199–206, 2009.

[10] K. Chantarasathaporn and C. Srisa-an. Object-oriented programming strategies in c# for power conscious system. *International Journal of Computer Science {Online}*, 1(1), 2006.

[11] A. Chatzigeorgiou and G. Stephanides. Energy metric for software systems. *Software Quality Control*, 10(4):355–371, 2002.

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

[13] K. D. Cooper, D. Subramanian, and L. Torczon. Adaptive optimizing compilers for the 21st century. *J. Supercomput.*, 23(1):7–22, 2002.

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

[15] Y. Fei, S. Ravi, A. Raghunathan, and N. K. Jha. Energy-optimizing source code transformations for operating system-driven embedded software. *ACM Trans. Embed. Comput. Syst.*, 7(1):1–26, 2007.

[16] S. V. Gheorghita, H. Corporaal, and T. Basten. Iterative compilation for energy reduction. *J. Embedded Comput.*, 1(4):509–520, 2005.

[17] W. Gilani and O. Spinczyk. Dynamic Aspect Weaver Family for Family-based Adaptable Systems. In *Proceedings of Net.ObjectDays*, pages 94–109. Gesellschaft für Informatik, 2005.

[18] Global Action Plan. An inefficient truth (white paper), 2007. http://greenict.org.uk/sites/default/files/An%20Inefficient%20Truth%20-%20Full%20Report.pdf.

[19] A. Grover. Modern system power management. *Queue*, 1(7):66–72, 2003.

[20] I. Hong and M. Potkonjak. Power optimization in disk-based real-time application specific systems. In *Proceedings of the International Conference on Computer-aided design*, pages 634–637. IEEE Computer Society, 1996.

[21] T. Ishihara and H. Yasuura. Voltage scheduling problem for dynamically variable voltage processors. In *Proceedings of the International Symposium on Low power electronics and design*, pages 197–202. ACM, 1998.

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

[23] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. V. Lopes, J.-M. Loingtier, and J. Irwin. Aspect-Oriented Programming. In *Proceedings of the European Conference on Object-Oriented Programming (ECOOP)*, volume 1241 of *Lecture Notes in Computer Science*, pages 220–242. Springer-Verlag, 1997.

[24] P. Korteweg, A. Marchetti-Spaccamela, L. Stougie, and A. Vitaletti. Data aggregation in sensor networks: Balancing communication and delay costs. *Theor. Comput. Sci.*, 410(14):1346–1354, 2009.

[25] M. S. Lane, A. Howard, and S. Howard. The energy inefficiency of office computing and potential emerging technology solutions. *Journal of Issues in Informing Science & Information Technology*, 6:795–808, 2009.

[26] Y. Li and J. Henkel. A framework for estimating and minimizing energy dissipation of embedded hw/sw systems. pages 259–264, 2002.

[27] A. Nicoara, G. Alonso, and T. Roscoe. Controlled, systematic, and efficient code replacement for running java programs. *SIGOPS Operating Systems Review*, 42(4):233–246, 2008.

[28] M. Pedram. Power optimization and management in embedded systems. In *Proceedings of the Asia and South Pacific Design Automation Conference*, pages 239–244. ACM, 2001.

[29] T. Pering, T. Burd, and R. Brodersen. The simulation and evaluation of dynamic voltage scaling algorithms. In *Proceedings of the International Symposium on Low power electronics and design*, pages 76–81. ACM, 1998.

[30] P. Pillai and K. G. Shin. Real-time dynamic voltage scaling for low-power embedded operating systems. In *Proceedings of the ACM Symposium on Operating Systems Principles*, pages 89–102. ACM, 2001.

[31] J. Pisharath, A. Choudhary, and M. Kandemir. Reducing energy consumption of queries in memory-resident database systems. In *Proceedings of the International Conference on Compilers, architecture, and synthesis for embedded systems*, pages 35–45. ACM, 2004.

[32] C. Poellabauer, T. Zhang, S. Pande, and K. Schwan. K.: An efficient frequency scaling approach for energy-aware embedded real-time systems. In *Proceedings of the International Conference on Architecture of Computing Systems*, 2005.

[33] M. Poess and R. O. Nambiar. Energy cost, the key challenge of today's data centers: a power consumption analysis of tpc-c results. *Proc. VLDB Endow.*, 1(2):1229–1240, 2008.

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

[35] 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.

[36] S. Rivoire, M. A. Shah, P. Ranganathan, and C. Kozyrakis. Joulesort: a balanced energy-efficiency benchmark. In *Proceedings of the International Conference on Management of Data*, pages 365–376. ACM Press, 2007.

[37] E. Saxe. Power-efficient software. *Queue*, 8(1):10–17, 2010.

[38] V. Tiwari, S. Malik, and A. Wolfe. Compilation techniques for low energy: An overview. In *Proceedings of Symposium on Low Power Electronics*, pages 38–39, 1994.

[39] H. Tomiyama, T. Ishihara, A. Inoue, and H. Yasuura. Instruction scheduling for power reduction in processor-based system design. In *Proceedings of the International Conference on Design, automation and test*, pages 855–860. IEEE Computer Society, 1998.

[40] M. A. Viredaz, L. S. Brakmo, and W. R. Hamburgen. Energy management on handheld devices. *Queue*, 1(7):44–52, 2003.

[41] T. Yokoyama, G. Zeng, H. Tomiyama, and H. Takada. Analyzing and optimizing energy efficiency of algorithms on dvs systems a first step towards algorithmic energy minimization. In *Proceedings of the Asia and South Pacific Design Automation Conference*, pages 727–732. IEEE Press, 2009.

Assessment of Product Derivation Tools in the Evolution of Software Product Lines: An Empirical Study

Mario Torres[1], Leopoldo Teixeira[2], Elder Cirilo[3], Uirá Kulesza[1],
Matheus Sousa[1], Paulo Borba[2], Thais Batista[1], Rosana Braga[4], Paulo Masiero[4], Carlos Lucena[3]

[1]Computer Science Department, Federal University of Rio Grande do Norte, UFRN, Brazil
[2]Informatics Center, Federal University of Pernambuco, UFPE, Brazil
[3]Pontifical Catholic University of Rio de Janeiro, PUC-Rio, Brazil
[4]Institute of Mathematical and Computer Sciences, University of Sao Paulo, USP, Brazil

{mario, uira, thais, matheus}@dimap.ufrn.br, {lmt, phmb}@cin.ufpe.br,
{ecirilo, lucena}@inf.puc-rio.br, {rtvb, masiero}@icmc.usp.br

ABSTRACT

Product derivation approaches automate the customization process of software product lines. Over the last years, many tools have been proposed aiming at synthesize and generate products from a set of reusable assets. These tools adopt different techniques and strategies to implement and automate the product derivation activities. In this paper, we analyzed six modern product derivation tools (Captor, CIDE, GenArch, MSVCM, pure::variants, XVCL) in the context of evolution scenarios of a software product line. Our study has adopted several metrics to analyze the modularity, complexity and stability of product derivation artifacts related to configuration knowledge along different releases of a mobile product line. The preliminary results of our study have shown that approaches with a dedicated model or file to represent the CK specification can bring several benefits to the modularization and stability of a software product line.

Categories and Subject Descriptors

D.2.8 [**Metrics**]: Product Metrics.

General Terms

Measurement, Experimentation

Keywords

Product Derivation Tools, Measurement

1. INTRODUCTION

A software product line (SPL) [8] is a set of related software systems from a particular market segment that share common functionalities, but are sufficiently distinct from each other. Existing approaches [9, 8] propose and motivate SPL development by means of the specification, modeling and implementation of features. A feature [9] is a system property or functionality that is relevant to a stakeholder. It is used to capture commonalities and discriminate variabilities among SPL systems. SPL development involves the design and implementation of core assets (components, frameworks, libraries and others) that adequately modularize common and variable features during domain engineering [9].

Product derivation [10] refers to the process of building a product from the set of code assets implemented for a SPL. It encompasses the selection, composition and customization of these code assets, in order to address a specific SPL product (configuration). Existing product derivation approaches [7, 6, 22, 4, 20, 13, 14, 3] automate the synthesis and customization of SPL products. Over the last years, many tools have been proposed with this aim. They adopt different techniques and strategies to implement and automate the product derivation activities, varying in different perspectives, such as: (i) from visual and model-based tools to textual and domain-specific approaches that specify the problem space (e.g., feature model), solution space (e.g., code assets) and configuration knowledge (mapping between features and code assets) from the SPL; and (ii) they adopt a positive or negative derivation process to customize and generate SPL products.

Many tools have been proposed, with several advantages of their adoption in real and industrial scenarios [7, 6, 22, 4, 20, 13, 14, 3]. However, there are few studies addressing the assessment and comparison of these tools that demonstrates the real impact, benefits and disadvantages of using a specific tool. Existing research work focuses on qualitative tool analysis [18, 21, 16, 15]. None of the existing studies have explored or analyzed the product derivation artifacts produced during the evolution of an existing SPL. Besides, to the best of our knowledge, there is no existing work that quantifies metrics that assess the modularity, complexity and stability of product derivation artifacts during the evolution of SPLs.

In this context, this work proposes to assess and compare existing product derivation approaches considering the evolution releases of a SPL. Our analysis focuses mainly on the modularity, complexity, and stability attributes of derivation artifacts specified to support the process of automatic product derivation. Existing metrics adopted in other recent empirical studies [12, 19] are adapted to quantify these attributes in product derivation artifacts that specify the configuration knowledge (CK) between features and code assets. The metrics are computed in the aspect-oriented implementations of four evolution releases of MobileMedia [12], a software product line for media (photo, video and audio) management on mobile devices. The following six existing product derivation tools are analyzed and compared in our study from the perspective of CK specification: Captor [20, 13], CIDE

[14], GenArch [7], Hephaestus [5], pure::variants [3] and XVCL [22]. The preliminary results of our study have shown that approaches with a dedicated model or file to represent the CK specification can bring several benefits to the modularization and stability of a software product line.

The remainder of this paper is organized as follows. Section 2 presents the study settings of our comparative study: it overviews the investigated product derivation approaches, it details the phases and assessment procedures adopted, and finally, it describes the metrics adopted to quantify the modularity, complexity and stability of the SPL releases. Section 3 analyzes and discusses the results obtained for the metrics considering the different releases of MobileMedia. Some general discussions are made on section 4. Section 5 relates our study to other existing ones developed by the community. Finally, Section 6 concludes the paper and provides directions for future work.

2. STUDY SETTINGS

This section presents detailed information about our assessment of product derivation tools. The main aim of our study was to analyze and observe the modularity and stability of existing product derivation artifacts, and also to validate the usefulness of some metrics in the quantification of these attributes. In Section 2.1, we present the approaches we have evaluated in this study. Section 2.2 presents the phases and assessment procedures of our preliminary study. Section 2.3 describes the metrics suite adopted to enable the modularity, complexity and stability analysis of the approaches.

2.1 Product Derivation Approaches

This section provides an overview of the evaluated approaches. We discuss their particularities, identifying how the configuration knowledge (CK) is expressed in each approach. In most of the approaches, **feature models** [9] are used to represent the commonalities and variabilities of the SPL, defining its scope. Feature models denote the set of products that can be generated for the SPL, through the relationships between features and their types (alternative, optional, mandatory, and so on). The approaches in our preliminary study were chosen according to the following criteria: (i) they represent approaches that were developed by our research groups (Captor, MSVCM and GenArch) or they are considered relevant product derivation tools in the research or industrial community (CIDE, pure::variants and XVCL); (ii) they are code-oriented, meaning that when we refer to **reusable assets**, we are mainly referencing code assets, such as classes, aspects, interfaces, packages, except stated otherwise; and (iii) there is an available implementation of the approach in order to allow its use in the case study. Unfortunately, due to time restrictions, it was not possible to include or consider other product derivation tools in this preliminary study. We intend to extend and consider new approaches in a future and more controlled study.

Captor. The Captor tool is a Configurable Application Generator used to support the development of applications on a specific domain [20]. It covers domain and application engineering. It uses an Application Modeling Language (AML), used in a way similar to a feature model, that can be specified through a graphical user interface, or directly in XML. The CK is also composed by templates with XSL tags, and a mapping file called rules.xml, that links the defined AML with the templates. Captor also provides pre and post processors, also specified on XML files, which can be used to define tasks like copying mandatory files to the output directory. The application engineer defines an instance of the AML previously created to build and derive a product.

CIDE. The Colored Integrated Development Environment (*CIDE*) is a SPL tool for decomposing legacy code into features [14]. It follows the paradigm of virtual separation of concerns, i.e., developers do not physically extract the feature code, but just annotate code fragments inside the original code, in a similar fashion to conditional compilation tags (*#ifdef*). However, instead of using tags in comments throughout the code, it uses background colors. So, code fragments belonging to a feature are shown with the background color of the feature. Another difference to conditional compilation is that annotations are disciplined, in order to prevent syntax and type errors. The underlying structure of the code to be annotated is considered, allowing developers to annotate, and thus, remove from assets, only program elements like classes, methods, or statements. These annotations represent the CK information, associating these elements to features.

GenArch. GenArch [7, 6] is a model-based tool for automating the product derivation process. The GenArch approach is centered on the definition of three models: feature, architecture and configuration. The architecture model defines a visual representation of the reusable assets in order to relate them to features. The configuration model is responsible for defining the mapping between features and assets, representing the CK. This model is fundamental to link the problem space (features) to the solution space (implementation assets), and to enable automatic product derivation.

MSVCM. The Modeling Scenario Variability as Crosscutting Mechanisms (*MSVCM*) approach [4] was initially proposed to deal with requirements variability, but it has been extended to deal with variabilities in source code and build files [5]. In MSVCM, the CK is specified into a separate model, relating features and their combinations (feature expressions) to transformations that translate SPL assets into product specific artifacts. The approach is named crosscutting because product derivation is resultant of a weaving process that takes as input artifacts such as the feature model, configuration knowledge, instance model, and so forth. These models crosscut each other with respect to the resulting product. If a given feature expression is evaluated as *True* for a given product (defined in the instance model), the related transformations are applied.

pure::variants. pure::variants [3] is a SPL model-based product derivation tool. Its modeling approach considers mainly two models: feature and family models. The family model describes the internal structure of the individual components and their dependencies on features. The family model is structured in several levels. The highest level is formed by the components. Each component represents one or more functional features of the solutions and consists of logical parts of the software (reusable assets). The physical elements can be assets that already exist, assets that will be created and transformations that will be performed based on the feature selection. A transformation can be any activity, such as copying code assets from a repository to a specific location, customizing configuration files, or even UML models.

XVCL. The XML-based Variant Configuration Language (*XVCL*) [22] is a language for configuring variability in textual-based assets. XVCL is based on Bassett's frames [2], every file on its structure needs to be a frame (XML file combined with code and XVCL commands) linked to other frames. This hierarchical structure is called *x-framework*. This approach has no specific structure to organize the CK. Instead, we use XVCL variables to set features and then validate if that feature should be present or not in the derivated product. The XVCL processor then, given a set of features (variables) and their values (selected or not) processes the frames files in order to generate the product.

2.2 Study Phases and Assessment Procedures

Our study was organized in the following major phases: (i) specification of the SPL artifacts related to product derivation considering all the approaches presented in Section 2.1; (ii) quantification of the selected metrics over the different derivation artifacts produced for each one of the investigated approaches; and (iii) quantitative analysis and assessment of the obtained results for the different modularity, complexity and stability metrics adopted in our study. Following we provide additional details of these phases.

In the first phase of our study, the aspect-oriented implementations of 4 releases (release 4 to 7) of MobileMedia SPL [12] were considered to implement the different artifacts of the product derivation approaches. The available documentation of MobileMedia was used as a base to specify and implement the derivation artifacts. MobileMedia (MM) was selected to be part of our study for different reasons. First, because it is an expressive SPL implemented with modern technologies, including an aspect-oriented language (AspectJ) and the Java Micro Edition (Java ME) API. Second, it has been used and validated in many other empirical studies [12, 1]. Finally, MM provides different evolution releases which allowed us to observe the effects of change scenarios along derivation artifacts considering the different approaches. This last criterion was preponderant to the choice of MM to this preliminary study.

During the specification of the derivation artifacts, we established alignment rules between the approaches in order to guarantee that: (i) the best practices of each approach were used to implement the artifacts; and (ii) the comparison between the derivation artifacts was equitable and fair. Five researchers performed these alignment activities. All misalignments found were discussed between the study participants and eventual corrections were applied to the artifacts implementation to guarantee their alignment. It was ensured, for example, that: (i) the same set of common and variable features were used in the derivation artifacts considering each release of MM; (ii) every variability and implementation artifacts were expressed using the appropriate mechanisms of the product derivation approaches; and (iii) the CK specifications in the different approaches are consistent between them, which means that all product derivation approaches are specifying the same products that can be automatically produced from each MM release.

After implementing MM derivation artifacts using the six different approaches, we applied and quantified the modularity, complexity and stability metrics along these different artifacts. We considered the artifacts that are responsible to specify the configuration knowledge in the different approaches. Our main aim was to quantify the tangling, scattering, size and instability of the derivation artifacts in the light of change scenarios demanded by MM evolution. Additional details about the adopted metrics to quantify these properties are presented in the next section. Finally, after the collection of all the metrics, the computed data was organized in spreadsheets and graphics in order to be analyzed. Results of this analysis are presented in Section 3.

2.3 The Metrics Adopted in Our Study

In order to compare the CK specification of the different product derivation approaches, we have selected a metrics suite to enable their quantitative analysis. The metrics are divided into three main groups: (i) modularity, (ii) complexity and (iii) stability.

Modularity. The modularity metrics are adapted from previously proposed metrics by Sant'Anna et al [19]. This previous work has proposed a set of modularity metrics to measure the separation of concerns in aspect-oriented implementations. The main goal of these metrics is to quantify the degree of scattering and tangling of concerns in aspect-oriented artifacts. They have been used and validated not only in the assessment of aspect-oriented implementations, but also to artifacts produced in other development stages: such as requirements, architecture and design, textual and model specifications [4,12]. In our study, we have adapted these metrics to quantify the scattering and tangling of CK specifications along product line assets (configuration files, derivation models, templates and source code) that are implemented to enable automatic product derivation.

We measure the scattering counting 1 for each SPL asset that contains some sort of CK specification, including any textual document or model associated with CK, when applied. In the CIDE approach, we counted every code asset colored according to a specific feature, as a unit of CK scattering. Tangling is calculated in a similar way than the Concern Diffusion over Lines of Code (CDLoC) metric, where we count the number of concern switches in a given source code asset [19]. In our case, we calculate this metric considering the CK specification as the concern. Thus, for each derivation artifact that includes some sort of CK specification tangled with specification or code related to another concern (variabilities and implementation), there is a switch, which we count. If the approach has a dedicated CK model or textual specification, we do not take this model into account for this metric, because it completely modularizes this concern.

Complexity. The goal of this analysis is to measure the effort needed by domain engineers to prepare the artifacts that support the automatic product derivation in a specific approach. Complexity is directly related to the size of configuration items needed to represent the CK in each approach. Two metrics were used: (i) number of tokens in CK sentence expressions; and (ii) number of CK sentence expressions.

The first metric counts the number of tokens needed to build the CK, i.e., the data that the domain engineer must effectively write to configure the SPL according to the feature model. The counting was based on the native tokens provided by each language/tool to represent the CK. In the XVCL approach, for example, we define the following expression for the feature Photo: `<select option="Photo"> <option value="yes">`. Here we count the total of 16 tokens, 8 for each statement. On the other hand, in the GenArch tool, it is only necessary to provide the feature name associated - in this case, *Photo* - with a specific code asset in the configuration model. Thus the number of token in this case for GenArch was 1, because that is all the domain engineer has to write. The complete specification is transparent and is maintained through the configuration model.

The second metric, number of CK sentence expressions, analyzes the conciseness and expressiveness of the CK in the different approaches. It quantifies the amount of CK sentence expressions needed to support the product derivation process. When the SPL evolves, the concision of these expressions becomes even more important. In some approaches, it enables adding new assets without requiring the inclusion of a new expression. We count the number of CK sentence expressions by quantifying the amount of feature expressions specified in the CK artifacts. Note that this counting is independent of the effort to build the expression or its size. These are addressed by the number of tokens in CK sentence expressions metric.

Stability. The stability metric is used to analyze the impact of evolving the SPL on the derivation artifacts (CK). The metric was computed in terms of CK sentence expressions added, changed or removed during the SPL evolution. We measure the difference between releases, it is inspired on Yau and Collofello study [23]. A CK sentence expression is considered new when there is a new feature expression in the CK for that release. When a new code asset is included in the SPL, this not necessarily imply in the inclusion of a new CK feature expression. This depends on the approach. In some cases, an existing expression can be modified to address the new code assets. With this metric analysis we can measure the effort needed to evolve the CK during SPL maintenance.

3. STUDY RESULTS

In this section, we present and discuss the collected results for the modularity, complexity and stability metrics from our study. Our analysis considers the specification of the derivation artifacts considering 4 different releases of the Mobile Media SPL.

3.1 Modularity Analysis

The main goal of modularity analysis is to quantify the degree of scattering and tangling of the CK over the SPL assets.

Configuration Knowledge Scattering. We measured the degree of scattering of the configuration knowledge by quantifying all SPL assets that have some sort of CK specification in them. Source code, for example, might contain such information in the form of a conditional compilation tag (*#ifdef*). Figure 1 shows the results of the collected values for this metric considering the four releases of MobileMedia specified using the different approaches. We can observe that GenArch, pure::variants and MSVCM tools presented more stable and lower values for this metric compared to other approaches, even when new features and assets are added. This happens mainly because these approaches provide a separate model or configuration model to specify the configuration knowledge with the mapping between features and code assets. The CIDE, Captor, and XVCL approaches presented higher values for the CK scattering, as shown in Figure 1. CIDE presents a higher scattering because it does not provide dedicated support to modularize the CK. In this approach, every colored element in the code assets can be seen as a CK sentence expression that relates the element to a feature. The Captor and XVCL approaches also presented high values for the CK scattering metric. The product derivation assets from these approaches are composed by a set of specific files that describe transformation rules, which represent a significant part of the CK specification. Nonetheless, code assets also contain CK specification in the form of tags. This explains why these two approaches got quite similar values for CK scattering metrics in all MobileMedia releases. As we see in Figure 1, the collected values for the CK scattering metric were higher for the Captor and XVCL approaches.

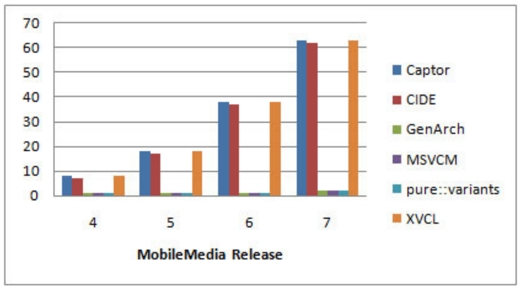


Figure 1. Configuration Knowledge Scattering.

Figure 7 illustrates the product derivation artifacts specified for five approaches that are responsible for the inclusion of the Photo and Music features in the MobileMedia SPL. For example, XVCL requires that each SPL asset that is processed to be a frame. In order to transform an asset in a frame, we have to include basic tags with some parameters, whether this frame has fine-grained variability or not. In the right side of Figure 7 for example, PhotoAndMusicAspect.aj.xvcl represents the correspondent aspect specified as a frame. On the other hand, Captor also needs to use XSL tags, and every variable asset has to be a template. The right side of Figure 7 also shows PhotoAndMusicAspect.aj aspect specified with a set of XSL tags used by Captor derivation artifacts. The left side of Figure 7 illustrates the configuration files of the Captor and XVCL approaches that are used to specify the transformation rules mentioned above. The scattering of these transformation rules and code tags along product derivation artifacts and code was quantified by the degree of scattering metric. Note that GenArch, MSVCM and pure::variants does not contain CK information on the code asset. The CK is modularized into a single file.

Configuration Knowledge Tangling inside Code Assets. Figure 2 shows the collected values for CK tangling metric. The Captor and XVCL approaches presented higher values for the tangling metric. This mainly occurs because both approaches must contain specific headers and footers in any code asset associated with CK. This information is used during the product derivation process of each approach. Figure 7(right side) shows, for example, that the PhotoAndMusicAspect.aj aspect needs to include configuration tags for both Captor and XVCL approaches. All the code assets associated with variabilities required the insertion of these same configuration tags thus contributing to improve the CK scattering. Precisely because GenArch, MSVCM and pure::variants, have the CK properly insulated, there is no CK tangling inside code asset.

The CK tangling metric was quantified in CIDE by the occurrence of color sentences spread along code assets. Because of that, CIDE initially presented a low tangling considering the MobileMedia release 4, but as the number of features increased, values got higher compared to other approaches. It mainly happens because with the increase of features to be managed, feature expressions also become more complex. GenArch, pure::variants and MSVCM had equivalent and the best results for the CK tangling metric considering all the releases. On the first three releases investigated (releases 4 to 6), the CK tangling for these approaches was zero. This occurred because the CK was adequately modularized into a dedicated model.

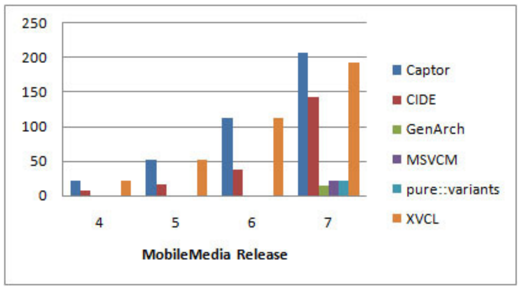


Figure 2. CK Tangling inside Code Assets.

We can also observe that fine-grained variability increases CK tangling in code assets. The fact that most of MobileMedia variations are well modularized with aspects contributed for the lower values of the scattering and tangling metrics in releases 4-6. However, release 7 has a particular case where one asset (OptionalFeatureAspect.aj) contains fine-grained variability that cannot be handled using only the CK model. Figure 3 illustrates this aspect, where the arguments passed on the declare precedence clause are variabilities and depend on the selection of specific features to be included in the aspect code. Because of that, this file needs to be processed using conditional compilation tags (MSVCM and pure::variants) or template processing (GenArch and Captor). Both techniques are used during product derivation to decide if part of this asset will be included or not. This kind of fine-grained variability usually happens on legacy SPLs[14]. Note that all the approaches and metrics from this study suffer influence of this fine-grained variability. This fine-grained variability was responsible for the light increase in the collected results for the CK tangling metric considering the Captor and XVCL for the release 7 of MobileMedia.

```
package lancs.mobilemedia.optional;
  ...
public aspect OptionalFeatureAspect {

 declare precedence : SMSAspect,
   CopyMultiMediaAspect, FavouritesAspect,
   SortingAspect,  PersisteFavoritesAspect;

}
```

Figure 3. OptionalFeatureAspect.

3.2 Complexity Analysis

The main goal of the two metrics on this group is to measure the complexity/size of the CK specification in each approach, and furthermore analyze how they behave when evolving the SPL.

Number of tokens in CK specification. The number of tokens metric allowed us to distinguish the size and complexity of CK specification on the different derivation approaches. Figure 4 shows the collected values for this metric. Captor is the approach that requires the higher number of tokens in CK specification. Reasons for that include the need for creating several CK decision expressions (task calls and definitions), and headers and footers specified in the code templates. XVCL also presented higher values for the number of tokens in CK metric. This happens due to the same reason of Captor, except that task definition and call is specified directly in XVCL files. In MSVCM, values for this metric are higher than GenArch since it is necessary to specify asset names and mandatory associations as well.

Figure 7 shows how we count tokens in all approaches (except CIDE), for the case of the PhotoAndMusicAspect asset. This aspect is related to the joint selection of the Photo and Music features. For this metric, in XVCL, Captor, GenArch, MSVCM and pure::variants approaches, we count 94, 129, 3,3 and 15, respectively. In the CIDE approach, since we just color code elements with associated features, there is no textual CK specification. Therefore, the metric values for all releases are zero. GenArch, pure::variants and MSVCM have values much smaller than other approaches. In GenArch, the architecture model that abstracts all the code assets is built automatically by the tool. Product line engineers only need to write feature expressions, associating features to assets. So, metric values tend to be lower for GenArch. This happens similarly in MSVCM.

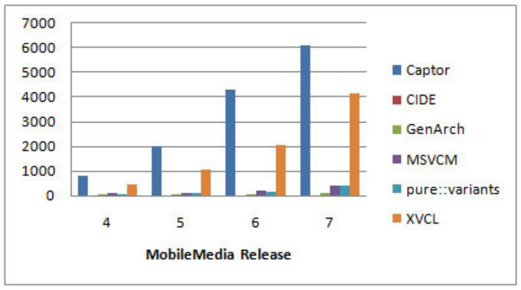
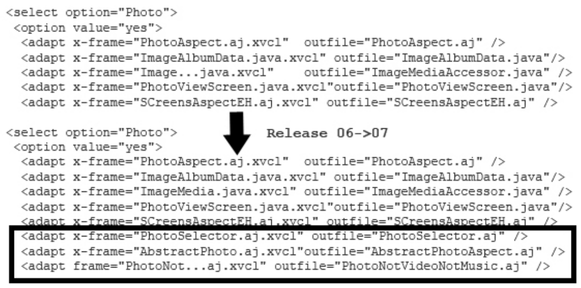


Figure 4. Number of Tokens in CK specification.



```
<select option="Photo">
 <option value="yes">
  <adapt x-frame="PhotoAspect.aj.xvcl"  outfile="PhotoAspect.aj" />
  <adapt x-frame="ImageAlbumData.java.xvcl" outfile="ImageAlbumData.java"/>
  <adapt x-frame="Image...java.xvcl"    outfile="ImageMediaAccessor.java" />
  <adapt x-frame="PhotoViewScreen.java.xvcl"outfile="PhotoViewScreen.java"/>
  <adapt x-frame="SCreensAspectEH.aj.xvcl" outfile="SCreensAspectEH.aj" />

<select option="Photo">                    Release 06->07
 <option value="yes">
  <adapt x-frame="PhotoAspect.aj.xvcl"  outfile="PhotoAspect.aj" />
  <adapt x-frame="ImageAlbumData.java.xvcl" outfile="ImageAlbumData.java"/>
  <adapt x-frame="ImageMedia.java.xvcl" outfile="ImageMediaAccessor.java" />
  <adapt x-frame="PhotoViewScreen.java.xvcl"outfile="PhotoViewScreen.java"/>
  <adapt x-frame="SCreensAspectEH.aj.xvcl" outfile="SCreensAspectEH.aj" />
  <adapt x-frame="PhotoSelector.aj.xvcl" outfile="PhotoSelector.aj" />
  <adapt x-frame="AbstractPhoto.aj.xvcl"outfile="AbstractPhotoAspect.aj" />
  <adapt frame="PhotoNot...aj.xvcl" outfile="PhotoNotVideoNotMusic.aj" />
```

Figure 5. XVCL – Sentence Expression.

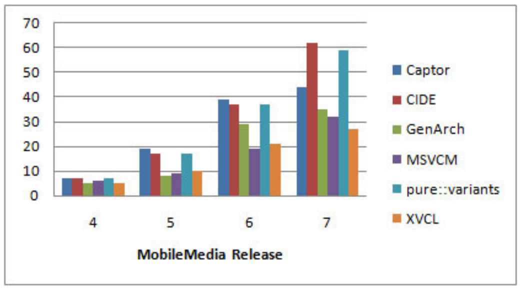
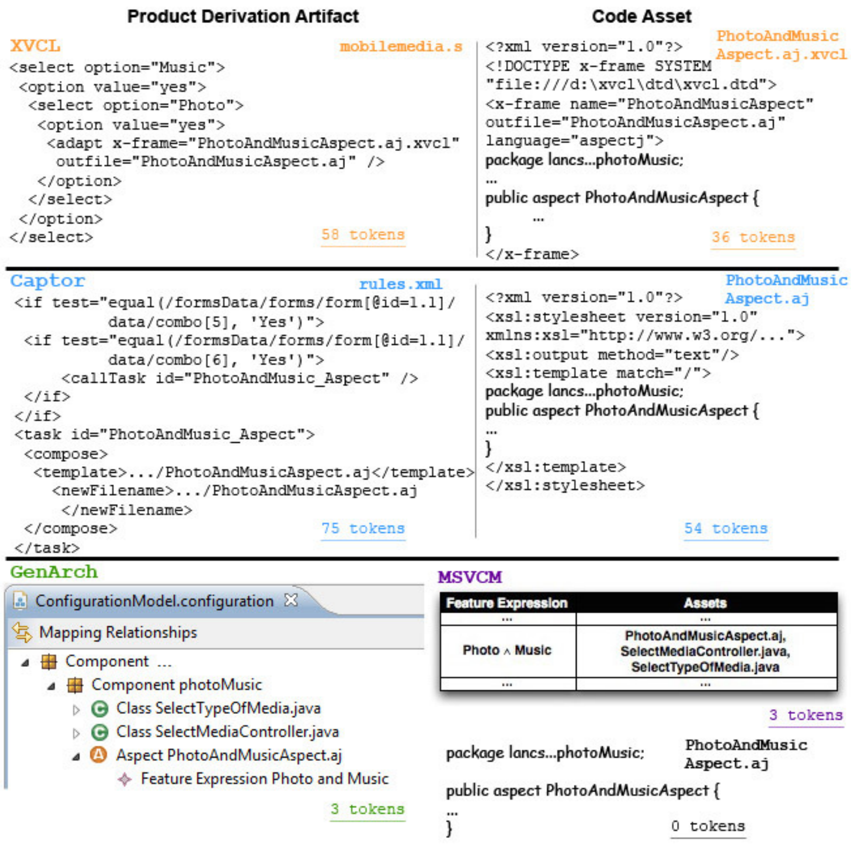


Figure 6. Sentence Expressions in CK Specification.

Sentence Expressions in CK specification. Figure 6 shows the number of sentence expressions for the different MobileMedia releases specified in the six approaches. Similarly to the tangling


Product Derivation Artifact
Code Asset
XVCL
mobilemedia.s
PhotoAndMusic Aspect.aj.xvcl
58 tokens
36 tokens
Captor
rules.xml
PhotoAndMusic Aspect.aj
75 tokens
54 tokens
GenArch
MSVCM
3 tokens
PhotoAndMusic Aspect.aj
3 tokens
0 tokens

```
<select option="Music">
 <option value="yes">
  <select option="Photo">
   <option value="yes">
    <adapt x-frame="PhotoAndMusicAspect.aj.xvcl"
     outfile="PhotoAndMusicAspect.aj" />
   </option>
  </select>
 </option>
</select>
```

```
<?xml version="1.0"?>
<!DOCTYPE x-frame SYSTEM
"file:///d:\xvcl\dtd\xvcl.dtd">
<x-frame name="PhotoAndMusicAspect"
outfile="PhotoAndMusicAspect.aj"
language="aspectj">
package lancs...photoMusic;
...
public aspect PhotoAndMusicAspect {
      ...
}
</x-frame>
```

```
<if test="equal(/formsData/forms/form[@id=1.1]/
          data/combo[5], 'Yes')">
 <if test="equal(/formsData/forms/form[@id=1.1]/
          data/combo[6], 'Yes')">
     <callTask id="PhotoAndMusic_Aspect" />
 </if>
</if>
<task id="PhotoAndMusic_Aspect">
 <compose>
  <template>.../PhotoAndMusicAspect.aj</template>
    <newFilename>.../PhotoAndMusicAspect.aj
     </newFilename>
 </compose>
</task>
```

```
<?xml version="1.0"?>
<xsl:stylesheet version="1.0"
xmlns:xsl="http://www.w3.org/...">
<xsl:output method="text"/>
<xsl:template match="/">
package lancs...photoMusic;
public aspect PhotoAndMusicAspect {
...
}
</xsl:template>
</xsl:stylesheet>
```

ConfigurationModel.configuration
Mapping Relationships
- Component ...
  - Component photoMusic
    - Class SelectTypeOfMedia.java
    - Class SelectMediaController.java
    - Aspect PhotoAndMusicAspect.aj
      - Feature Expression Photo and Music

Feature Expression	Assets
...	...
Photo ∧ Music	PhotoAndMusicAspect.aj, SelectMediaController.java, SelectTypeOfMedia.java
...	...

```
package lancs...photoMusic;

public aspect PhotoAndMusicAspect {
...
}
```

Figure 7. Approaches and Metrics.

metric, the CIDE approach presented a considerable increase when adding new features. A sentence expression in CIDE is considered as a colored element. Therefore, these metric values are somewhat correspondent to the scattering metric values for the CIDE approach. The XVCL and MSVCM approaches can group many assets into a single sentence. A difference is that in MSVCM, we also specify the mandatory relationships as previously mentioned. Figure 6 illustrates this grouping characteristic of XVCL. It shows that, in release 6, the Photo feature is associated with 5 assets. In the next release, 3 new assets are added to this expression, and no other sentence needs to be created. In Captor, pure::variants and GenArch, we need to add 3 sentence expressions. This can be mitigated in pure::variants and GenArch by associating higher level abstractions, such as packages, with feature expressions. However, if feature implementation is highly scattered, it might not be possible to do so. In Figure 7 the sentence expressions are showed for the different approaches.

3.3 Stability Analysis

This analysis looks at three different perspectives of sentence expressions between releases: added, modified and removed.

Added. Figure 8 (a) shows the number of sentence expressions added in the configuration knowledge of each approach between releases. When adding code assets, GenArch, pure::variants, Captor and CIDE treat assets individually, so they have larger effort than MSVCM and XVCL, which can group these assets. In GenArch and Captor, the inclusion of an asset represents a new sentence expression added to the configuration knowledge.

Changed. Figure 8(b) shows the number of sentence expressions changed in the configuration knowledge. The releases 4, 5 and 6 had many new assets and few expressions, or none in most cases,

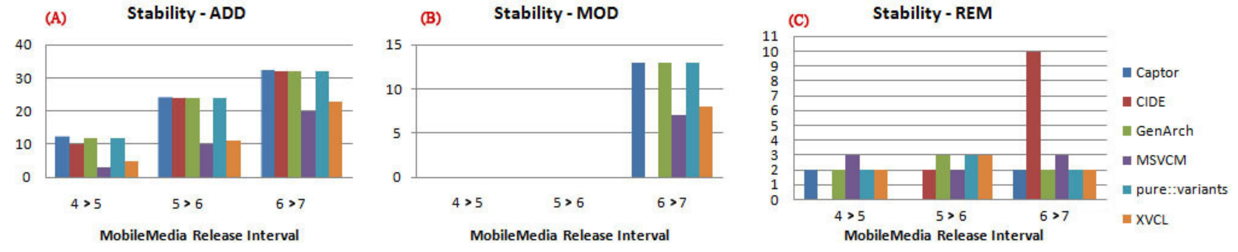


Figure 8. Stability – Added (A), Modified (B) and Removed (C) .

that could be reused. Again, the approaches (GenArch, Captor and pure::variants) that treat assets individually had larger effort than the other ones. The CIDE approach has not changed items, because when changes occur in an asset, you must remove its color and then mark the new artifact with this color.

Removed. Figure 8(c) shows the numbers of sentence expressions removed from the CK in each approach. This usually occurs when code assets are removed. These metric results are very similar to almost all approaches, variation happens only in cases that the approach could reuse some expressions, reflecting on Figure 8(b). The fact that CIDE cannot change the sentence expressions results in the increase of the number of removed items.

4. DISCUSSIONS

In this section, we discuss interesting issues and lessons learned related to the assessment of the product derivation approaches in our preliminary evolution study.

General Analysis of the Study Results. The GenArch, pure::variants and MSVCM approaches presented the best results considering most of the modularity and complexity metrics. The main reason that contributed to the good performance of these approaches is the use of a separated CK specification. Captor and XVCL obtained the less satisfactory results considering the modularity and complexity metrics. This mainly occurred due to the textual and complex nature of CK specification provided by these approaches. Considering initial MobileMedia releases, CIDE presented reasonable results for the modularity and complexity metrics. However, it did not scale along the subsequent ones in terms of CK scattering and tangling, because of the increasing number of CK color sentences specified in the code assets. Regarding the stability metrics, we have observed that Captor, GenArch, MSVCM and XVCL required to add and modify a small and stable number of CK sentence expressions along the different releases. MSVCM was the approach that presented the better stability for the CK expressions. On the other hand, CIDE exhibited higher values for the number of removed CK expressions due to the need to reassign new colors to CK sentences that are changed in the code assets along the releases.

Automated and Model-driven Tools. Our study has revealed that the CK specification in a dedicated model or file can bring several benefits to the modularization, complexity and stability attributes. That was the case for GenArch, MSVCM and pure::variants approaches. Captor and XVCL did not obtained better results in most of the modularization and complexity metrics. These approaches require manually written transformation rules related to the product derivation process. It is interesting to notice that the simple automated support to the specification and generation of the transformation rules that represents CK in both Captor and XVCL approaches could bring equivalent results to the other approaches.

CK Specification inside Code Assets. Another interesting issue that our study has also revealed is the positive/negative impact of the variability implementation technique in the modularization of the product derivation artifacts. Most of the variabilities implemented for MobileMedia releases were codified using aspect-oriented programming. This contributed to the positive values obtained for most of the modularity, complexity and stability for the different approaches. Analysis of the MobileMedia implementation using conditional compilation (CC), for example, would find a large number of CC directives (IFDEFs) spread and tangled along the different code assets. All these CC directives can also contribute to the instability of the product derivation artifacts during SPL evolution because they are not adequately modularized. Although the adoption of aspect-oriented programming in the MobileMedia implementation brought advantages and benefits to the specification of the product derivation artifacts, our study also illustrated that fine-grained variabilities encountered inside code assets can lead to difficult CK modularization scenarios, even for approaches that provides a separate CK specification. This was illustrated in our study by the OptionalFeatureAspect that establishes the precedence between aspects (variabilities) that will be applied to the SPL core.

5. RELATED WORK

To the best of our knowledge, we are not aware of empirical research on the assessment of different product derivation tools used in the evolution context for SPLs. Many works evaluate a single approach for SPL implementation used in a SPL evolution scenario. The original MobileMedia study [12] compares the negative and positive impact of using aspects for implementing SPLs, instead of conditional compilation. There are some similarities to our work, since we are using the same releases and adapting some of the used metrics. So, instead of evaluating the code only, in this work we focus on how different CK representations provided by the product derivation tools behave in a SPL evolution scenario.

Mannisto et al. discuss the problem of managing configuration knowledge evolution [17], describing key elements and presenting an outline for future work. They present an approach based on the Generic Product Structure Model and directions on how this approach should be used. However, they do not evaluate its use in a case study, as we focus on this work.

Dhungana et al. present a tool-supported approach for treating evolution in Model-based Product Line Engineering [11]. The approach decomposes large variability models into sets of interrelated model fragments. Such fragments are merged to provide full variability models. They report an experience of using it in an industrial collaboration. Some of the approaches we evaluate in this work (GenArch and pure::variants) are model-based and provide similar functionality. A difference to our evaluation is that due to model decomposition, they evaluate multi-team support for creating and maintaining SPL variability models. This is not something we investigated in this work, but it could be targeted in future work.

Many works compare product derivation tools in a systematic, but general way, not specifically focusing the CK. Rabiser et al. aim to identify and validate requirements for tool-supported product derivation [18]. Requirements are identified in a high-level way through a systematic literature review and validated through an expert survey. Sinnema and Delstra classify six variability modeling techniques [21] using a running example. They define a classification framework that lists a number of aspects that a variability modeling technique should possess. Lisboa et al. present a systematic review of domain analysis tools [16]. They focus on identifying (i) whether current available tools support a specific or generic process; (ii) their main functionalities; and (iii) the development context, and where it is being used --- academia or industry. Functionalities identified are categorized in priority levels: *essential*, *important*, and *low*. Khurum and Gorschek also conducted a systematic review of domain analysis solutions, but only for tools that focus on software product lines [15]. They focus on the usability and usefulness of the existing solutions, in terms of scalability of introduction and use, better alternative investment, and effectiveness. These works focus on comparing functionalities that existing product derivation tools provide. As mentioned, they do not focus on the configuration knowledge, nor evaluate tools in the evolution context. Khurum and Gorschek findings [15] indicate an absence of qualitative and quantitative results from empirical application of the approaches. Such absence, they argue, complicates the task of evaluating usability and usefulness of existing solutions. In this work, we attempt to fill this gap, through a quantitative and qualitative assessment of the configuration knowledge using different releases of the MobileMedia SPL.

6. CONCLUSIONS AND FUTURE WORK

This paper presented a preliminary empirical study that focused on comparing different product derivation approaches and their configuration knowledge representation. We performed this comparison through a quantitative analysis that measured modularity, complexity and stability attributes along product derivation artifacts produced for four evolution releases of a mobile SPL. Different metrics were used to quantify these attributes in terms of scattering, tangling, number of tokens and amount of CK sentence expressions, and number of added/removed/changed CK expressions considering all the SPL releases. We noticed that the adoption of specific approach strategies can lead to positive results in terms of the investigated attributes. From a general analysis, we concluded that approaches with a dedicated model or file to represent the CK specification had better results on many aspects. It was also observed that modest adaptations in some existing tools and approaches can bring a significant improvement on their performance, such as: (i) the automatic generation of headers and footers on Captor and XVCL tools; and (ii) the introduction of the capability to relate different sentence expression to two or more assets.

As a future work, we plan to replicate our study for different SPL domains and different variability implementation techniques in order to observe if the same obtained results of this study can be found in other SPL evolution scenarios. Besides, we also intend to include other existing product derivation tools in our future empirical studies. We intend to define and run more controlled experiments and case studies that follow the guidelines of the empirical software engineering. Last but not least, we are also working to adopt many of the findings and guidelines provided by our study in the design and implementation of our product derivation tools.

Acknowledgements. This work is supported in part by Brazilian Research Council (CNPq), grants 313064/2009-1 and 480978/2008-5 ; and CAPES/PROCAD, grant 090/2007.

7. REFERENCES

[1] J. Barreiros and A. Moreira. A Model-based Representation of Configuration Knowledge. In FOSD '09:, pages 43–48, New York, NY, USA, 2009. ACM.

[2] P. Bassett. Framing Software Reuse - Lessons from Real World. Prentice Hall, 1997.

[3] D. Beuche. Modeling and Building Software Product Lines with pure::variants. In SPLC, page 358. IEEE Computer, 2008.

[4] R. Bonifácio and P. Borba. Modeling Scenario Variability as Crosscutting Mechanisms. In AOSD'09, pages 125–136, USA.

[5] R. Bonifácio, L. Teixeira, and P. Borba. Hephaestus: A Tool for Managing Product Line Variabilities. In III SBCARS 2009 – Tools Session, pages 26–34, Natal, RN, Brazil, 2009.

[6] E. Cirilo, et al. Integrating Component and Product Lines Technologies. In H. Mei, editor, ICSR, volume 5030 of Lecture Notes in Computer Science, pages 130–141. Springer, 2008.

[7] E. Cirilo, et al. A Product Derivation Tool Based on Model-Driven Techniques and Annotations. J. UCS, 14(8):1344–1367, 2008.

[8] P. Clements and L. M. Northrop. Software Product Lines: Practices and Patterns. Professional. Addison-Wesley, 2001.

[9] K. Czarnecki and S. Helsen. Feature-based survey of model transformation approaches. IBM Systems Journal, 45(3):621–645, 2006.

[10] S. Deelstra, et al. Product derivation in software product families: a case study. Journal of Systems and Software, 74(2):173–194, Jan. 2005.

[11] D. Dhungana, T. Neumayer, P. Grunbacher, and R. Rabiser. Supporting evolution in model-based product line engineering. Proceedings of SPLC '08, IEEE Computer Society.

[12] E. Figueiredo, et al. Evolving Software Product Lines with Aspects: An Empirical Study on Design Stability. Proceedings of of ICSE'08, pages 261–270, New York, NY, USA, 2008. ACM.

[13] P. Junior and C. A. de Freitas. Geração de aplicações para linhas de produtos orientadas a aspectos com apoio da ferramenta Captor-AO, *MSc Dissertation*, University of São Paulo, Nov. 2008.

[14] C. Kastner, S. Apel, and M. Kuhlemann. Granularity in Software Product Lines. Proceedings of ICSE 2008, Leipzig, Germany, May 10-18, 2008, pages 311–320. ACM, 2008.

[15] M. Khurum and T. Gorschek. A Systematic Review of Domain Analysis Solutions for Product Lines. Journal of Systems and Software, 82(12):1982 – 2003, 2009.

[16] L. B. Lisboa, et al. A Systematic Review of Domain Analysis Tools. Information and Software Technology, 52(1):1 – 13, 2010.

[17] T. Mannisto, H. Peltonen, and R. Sulonen. View to product configuration knowledge modelling and evolution, Oct. 21 1996.

[18] R. Rabiser, et al. Requirements for Product Derivation Support: Results from a Systematic Literature Review and an Expert Survey. Information and Software Technology, 52(3):324 – 346, 2010.

[19] A. Garcia, et al. Modularizing Design Patterns With Aspects: A Quantitative Study. Proceedings of AOSD´2005, pp. 3-14, Chicago, 2005.

[20] E. K. Shimabukuro Junior. Um Gerador de Aplicações Configurável, *MSc Dissertation*, University of São Paulo, 2006.

[21] M. Sinnema, S. Deelstra. Classifying Variability Modeling Techniques. Information and Software Technology, 49(7):717 – 739, 2007.

[22] S. Swe, et al. XVCL: A Tutorial. In Proceedings of SEKE´2002, Ischia, Italy, 2002.

[23] S. S. Yau and J. S. Collofello. Design stability measures for software maintenance. IEEE Transactions on Software Engineering, 11(9):849–856, Sept. 1985.

Toolchain-Independent Variant Management with the Leviathan Filesystem*

Wanja Hofer[1], Christoph Elsner[1,2], Frank Blendinger[1], Wolfgang Schröder-Preikschat[1], Daniel Lohmann[1]

[1]Friedrich–Alexander University Erlangen–Nuremberg, Germany
[2]Siemens Corporate Research & Technologies, Erlangen, Germany

{hofer,elsner,wosch,lohmann}@cs.fau.de

ABSTRACT

Preprocessor-configured software needs tool support for the developer to be able to cope with the complexity introduced by optional and alternative code blocks in the source. Current approaches, which assist the software developer by providing preprocessed views, are all bound to a special integrated development environment. This eliminates them from being used both in industry settings (where domain-specific toolchains are often mandated) and in open-source projects (where diverse sets of editors and tools are being used and freedom of tool choice is crucial for the project success).

We therefore propose to tackle the problem at a lower level by implementing variant views at the *filesystem level*. By mounting one or more variants using our LEVIATHAN filesystem, we enable the use of standard tools such as syntax validators, code metric analysis tools, or arbitrary editors to view or modify a variant. The major benefit (and challenge) is the support for automatically writing back to the configurable code base when editing one of the mounted variant views.

Categories and Subject Descriptors

D.2.3 [**Software Engineering**]: Coding Tools and Techniques—*Program editors*; D.2.6 [**Software Engineering**]: Programming Environments; D.2.13 [**Software Engineering**]: Reusable Software

General Terms

Human Factors, Languages

*This work was partly supported by the German Research Council (DFG) under grants no. SCHR 603/4 and SCHR 603/7-1.

Keywords

Leviathan, Software Product Lines, Variability Implementation, Preprocessor-Based Configuration, Toolchain-Independent Variability Support, Filesystem Views

1. INTRODUCTION AND MOTIVATION

A lot of configurable software projects implement their variability in the sources using a preprocessor, which includes or excludes annotated code blocks depending on a given configuration. Preprocessor-based configuration is supported by all major software product line tools (e.g., pure::variants [1], Gears [7], etc.) and is especially prevalent in the domain of embedded systems and operating systems, because preprocessor-based configuration causes no run-time overhead. The matter of the fact, however, is that developers of systems such as Linux or eCos [11], an embedded configurable operating system, have to face a myriad of preprocessor directives and optional code blocks in the sources, even though they might only be working on the implementation of a single feature at a time. Thus, it has long been recognized that tool support is needed to cope with that complexity (colloquially termed #ifdef hell) to aid maintenance, evolution, and testing of such configurable software [6, 14]. For instance, in a previous study, we have found feature implementations in eCos to be highly scattered across different source files and to be tangled within the source files [9], rendering comprehension of certain files almost impossible.

Tools such as CIDE [6] or C-CLR [13] therefore each extend a special integrated development environment (IDE) and provide preprocessed views on the configurable code base depending on a given configuration. The main disadvantage of those approaches is that they force the developers into using that special IDE to cope with preprocessor complexity. This is infeasible both in industry projects, where toolchains are often fixed, and in open-source projects such as Linux, where the personal freedom of the developers to choose their editors and toolchains is of paramount importance[1]. Embedded software product lines, for instance, are developed in very heterogeneous setups: Engineers include domain experts in operating systems, in the actual embedded application, or specialized in drivers. Oftentimes, those engineers work in different companies supplying parts of the code, and they make use of different, special-purpose tools while developing and maintaining their subsystems, such as network analysis or real-time analysis tools. Most of those

[1]To put it bluntly: Kernel hackers hate Eclipse.

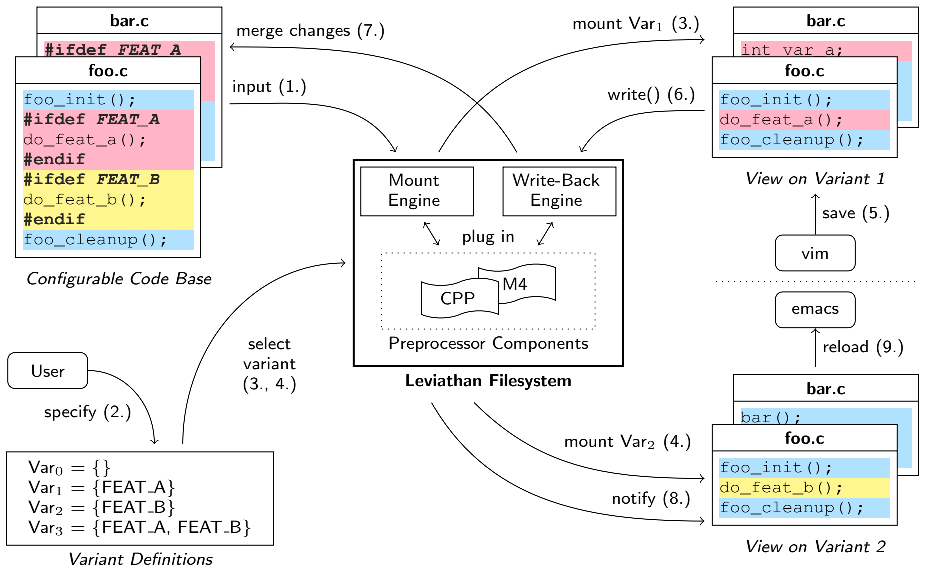


Figure 1: Example Work Flow in Leviathan.

tools are either unaware of—or even incapable of dealing with—configurable source code.

Unaware tools include debuggers, for instance, which show the complete configurable base code in a debug session although only one concrete variant is being debugged at a time, possibly obfuscating program comprehension due to #ifdef cluttering. Tools that are unaware of a source code base being configurable simply do not work too well on those code bases, or they do not work to their full potential. *Incapable* tools, on the other hand, *break* when they are fed configurable source code instead of stand-alone code. Such incapable tools include many kinds of source analysis tools such as for execution time analysis, call graph extraction, deadlock detection, syntax validators, reverse engineering tools that generate UML diagrams from source code, and others. Liebig et al., for instance, report that existing tool support for Java or C# is broken by CPP conditional compilation [8].

In order to better support configurable software projects in industry and the open-source community, we therefore propose to implement variant views at the filesystem level. Our LEVIATHAN filesystem is given a configurable software base with existing annotated code blocks (e.g., via CPP #ifdefs), and it can then mount views (i.e., it creates a virtual disk volume) depending on a user-provided configuration. The mounted views provide virtual directories and files, enabling the developer to use arbitrary file-based tools on that variant. Additionally, we aim at providing write-back support in LEVIATHAN, enabling the developer to directly edit the virtual variants; the changes are then automatically merged back into the configurable code base. By providing generic, toolchain-independent, and language-independent variant views, LEVIATHAN can therefore ideally support configurable software in industry and open-source projects.

In this paper, we first describe LEVIATHAN in an example work flow, complemented by a classification of use cases we envision our system to be used in (see Section 2). Then, we detail LEVIATHAN's architecture and implementation in Section 3 before providing a comprehensive description of its challenging write-back feature (see Section 4). After that, we discuss our approach and related work in Section 5 before concluding with Section 6.

Without loss of generality, for the remainder of the paper we assume the preprocessor directives to be #ifdefs and the preprocessor to be the subset of the C preprocessor CPP that is used for conditional compilation. The LEVIATHAN architecture, however, incorporates a plug-in mechanism to support arbitrary preprocessors (see Section 3).

2. WORK FLOW AND USE CASES

Figure 1 shows an example work flow how a target developer would use LEVIATHAN for software maintenance. First, he localizes a given configurable code base that he wants to reason about or work on (e.g., the Linux kernel sources) in the base filesystem. Second, he defines one or more variants as sets of enabled and disabled features (e.g., #define directives). Both of those pieces of information are fed into LEVIATHAN as input (steps 1 and 2 in Figure 1). The developer can then mount several variants simultaneously to different mount points by specifying the variant names (steps 3 and 4). After that, the user can operate as usual on the virtual directories and files, which are in fact slices of the original configurable code base. Operation includes read-only tasks such as reasoning about variants by viewing the differences between them as well as editing the virtual files with arbitrary tools; LEVIATHAN will merge back the changes into the configurable code base transparently in the background.

The work flow just described is, however, only one possible setting in which LEVIATHAN can come in handy. We envisage four types of settings, differing in whether the actual user is *human* or a *software tool*, and whether *read-only*

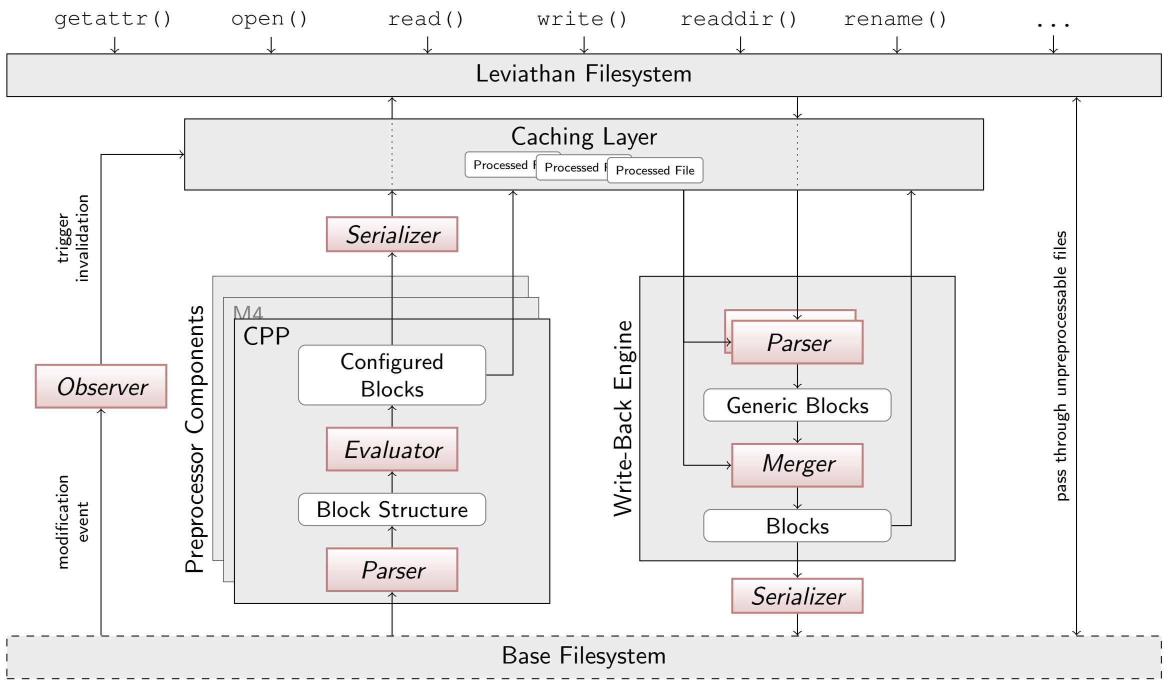


Figure 2: Leviathan Architecture and Data Flow.

or also *write-back* support is necessary. Each of the following four use cases provides an example for such a usage setting:

- **WCET analysis**: A real-time analysis tool shall be used to calculate the worst case execution time of a specifically configured variant (*user is a tool, read-only access*).
- **Code reasoning**: A software developer wants to get an understanding of the source code; the code of features irrelevant for the main functionality shall be excluded to improve comprehensibility (*user is human, read-only access*).
- **Feature refactoring**: A source code refactoring tool (e.g., Coccinelle [12]) shall be applied to a certain subset of features within the code base (*user is a tool, write-back support required*).
- **Maintenance changes**: The software developer fixes localized bugs in a configured variant and wants them to be merged back to the original source code base (*user is human, write-back support required*).

The different settings result in different general requirements for LEVIATHAN. First, if a human user is involved, configurable display options help to comprehend the source code independent of the capabilities of the employed editor. In some cases, marking the beginning and end of each feature block with a dedicated marker may hinder readability, whereas, in other cases, such markers are crucial to understand the prerequisites for a piece of code to be included. Second, LEVIATHAN's write-back support must prevent or handle cases of ambiguity when merging changes back to the configurable code base. Depending on the fact whether the user is human or a software tool, one strategy or the other will be more appropriate for disambiguation. We will discuss the corresponding write-back approaches in Section 4.

3. LEVIATHAN ARCHITECTURE AND IMPLEMENTATION

In this section we will describe LEVIATHAN's architecture, its implementation and present some preliminary evaluation results.

3.1 Architecture and Implementation

Internally, LEVIATHAN has a modular architecture and several layers, as depicted in Figure 2.

The topmost filesystem layer communicates with applications such as editors via the standard Linux VFS filesystem interface, which includes system calls such as `open()`, `close()`, `read()`, and `write()` to be called on files. This layer is implemented as a driver for FUSE [4], a framework that allows its actual drivers to run in user space; only a very small FUSE kernel module is executed in privileged mode. Thus, the LEVIATHAN filesystem driver can link to any third-party libraries built to support application development. FUSE supports various Unix variants as well as Mac OS X. There are also projects aiming at porting FUSE to Windows, making it the most portable framework for implementing filesystems.[2]

When processing a `read()` request, LEVIATHAN first determines if the corresponding file needs to be preprocessed

[2]See `http://sourceforge.net/apps/mediawiki/fuse/index.php?title=OperatingSystems`.

depending on its file type. Read requests on binaries, for instance, are directly passed through to the base filesystem. If a file *does* have to be preprocessed, Leviathan directs that request to its cache, which holds contents and meta data about files that have been processed before. An additional observer component monitors modification events on the base filesystem (via the `inotify` Linux kernel subsystem) and invalidates the corresponding cache entry appropriately upon changes. This is needed to synchronize both with changes made directly to the configurable code base on the base filesystem and with indirect changes to that code base via another mounted Leviathan variant (see also steps 6–8 in Figure 1). Only when a cache miss occurs does Leviathan direct the `read()` request to the corresponding preprocessor component.

The preprocessor component itself has a well-defined interface concerning the block structure it has to output for Leviathan to work on; it effectively encapsulates the syntax parsing and expression evaluating for a concrete preprocessor, such as CPP or M4 [10]. When the content of a virtual Leviathan file is requested for the first time or needs to be recalculated, the preprocessor component reads the base file from the base filesystem and uses its parsing subsystem to delineate configuration blocks encapsulated by preprocessor directives. The parser of our CPP preprocessor component, for instance, uses the Boost::Wave lexer [2] for that purpose, and it will only resolve those macros that are used for conditional compilation; that is, #defines used for definition of constants or #includes are *not* resolved in order not to impair code comprehension of the mounted variant. Each configuration block is bound to a preprocessor expression, which is stored together with the corresponding block. An expression evaluator then accesses the configuration that was handed to Leviathan at mount time and uses those preprocessor variable definitions (e.g., #defines in the CPP language) to evaluate if a given block is to be included in the virtual file or not. Note that this configuration can in fact be the output of an external feature modeling tool, which assures correctness in terms of feature selection and met dependencies.

Our CPP component parses the CPP expression and evaluates them. It supports some basic arithmetical operations as the original C preprocessor does. As a special feature, logical expressions are evaluated using three-valued logic; CPP variables and whole expressions can evaluate to *true*, *false*, or *undecided*. True blocks are included in the virtual file, false blocks are excluded, and undecided blocks are output together with their preprocessor annotations. As the negation of *undecided* also evaluates to *undecided*, the #else clause of an #if/#else statement evaluating to *undecided* is also included. This way, the developer can explicitly express partial configurations by setting features to undecided, besides being able to activate and deactivate features.

The data structure as output by the preprocessor component is then stored in Leviathan's cache to serve future read requests. Additionally, a serializer component computes the plain data stream that the application that has issued the `read()` call will be given as a result.

3.2 Preliminary Evaluation

Our preliminary evaluation of the Leviathan filesystem has yielded promising results. We have tested its performance by measuring the time required to read, parse, and output the complete source tree of Linux (version 2.6.31) and the eCos embedded operating system (CVS-version 2010-03-29). The test system has an Intel Core 2 Quad CPU Q9550 processor clocked at 2.83GHz, equipped with 4GB of RAM.

For Linux, the time to read, preprocess, and output (to `/dev/null`) its complete source tree of 408MB takes Leviathan 130 seconds. Directly reading and outputting the source tree without employing Leviathan (and therefore without preprocessing) took 14 seconds. Thus the slow down factor as caused by Leviathan is about 10. As Leviathan only parses the actually accessed files and we expect most use cases for Leviathan to involve only a rather small number of files (a human user, for example, only can read one file at a time), we do not consider this decrease to be a show stopper. Furthermore, both the 130 and the 14 seconds were produced without caching to ensure comparable figures. When using caching (the operating system's file system caching as well as Leviathan's caching), the figures decrease considerably, to 12 seconds for Leviathan and to 1 second for direct reading. The fact that Leviathan is still notably slower is caused by its implementation as a FUSE filesystem in user space, which by design causes expensive additional context switching overhead between kernel and user space.

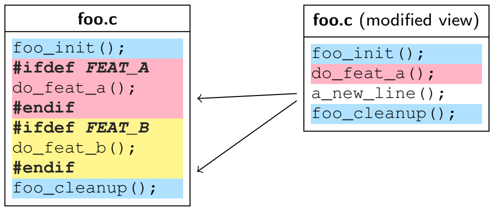
When using Leviathan to read, preprocess, and output the eCos embedded operating system, which has a code base of only 1MB, all figures drop well below 1 second and no noticeable disruptions occur in the work flow of a user. Although, in its current state of development, Leviathan is not optimized for speed, we consider its performance sufficient for the aspired use cases described in Section 2.

4. WRITE-BACK SUPPORT

In addition to offering read-only views for analysis tools and tasks that work on a configured variant (e.g., code metric tools or execution time analysis tools), we want to enable the software developer and his tools to *edit* a mounted variant. For instance, if he wants to perform maintenance changes (see also the corresponding use case in Section 2), this provides him with the ability to specify a configuration, mount the variant, debug it, and modify the variant code directly to get rid of the bug, eventually saving the changes in his editor (step 5 in Figure 1). In the background, the Leviathan filesystem will handle the write request by the editor appropriately by directing it to its write-back engine (step 6 in Figure 1; see also the corresponding architecture part in Figure 2).

The actual write-back support is challenging. To this end, Leviathan needs the additional information that the preprocessor component has computed—the mapping from *source configuration blocks* (source code blocks enclosed in preprocessor directives) to *variant blocks* (those configuration blocks actually visible in a given configuration) including their positions. Note that in the mounted view, changes in the configuration block *structure* are not supported. That is, if additional #ifdef blocks need to be introduced, for instance, this has to be done directly in the configurable code base.

Since Leviathan is editor-independent, it does not have an actual edit protocol available that shows which lines were changed in which manner. Instead, it needs to operate on discrete file content snapshots provided only when the



```
foo_init();
#ifdef FEAT_A
do_feat_a();
#endif
#ifdef FEAT_B
do_feat_b();
#endif
foo_cleanup();
```

```
foo_init();
do_feat_a();
a_new_line();
foo_cleanup();
```

Figure 3: Example Ambiguity Problem During Write-Back.

`write()` system call is issued (by saving the changes in an editor, for instance). Such discrete snapshots make it impossible to know for certain what the user actually did and *semantically* intended to express. Consider, for instance, which block to assign a line to that was inserted exactly between two variant blocks (see Figure 3). This example is but one of several edit scenarios that causes potential ambiguity in the write-back process. We have therefore found two different ways to deal with write-back requests.

4.1 Write-Back Heuristics

Using heuristics to merge changes performed on a mounted variant view back into the configurable code base is feasible as long as two conditions hold:

1. The configuration blocks in the configurable code base are rather large and therefore there are relatively few edges between variant blocks and a lot of context for merge algorithms to work with. A recent survey covering 40 software product lines suggests that this is a valid assumption [8].

2. The changes in the mounted view are saved rather incrementally and therefore the change sets are relatively small. Again, in that case, the merge algorithms have a lot of unchanged lines serving as context to work with. Maintenance changes would be a typical use case that fulfills this property (see also Section 2).

If those assumptions hold, then LEVIATHAN's write-back algorithm will provide proper merge results to be written back to the configurable code base. If one of those conditions *does not* hold, then it is LEVIATHAN's heuristics that will make a decision in corner cases, such as which block to assign an inserted line to, as described above and in Figure 3. In any case, if the code base itself is checked into a revision control system, then the merge results can still be double-checked in a difference view before actually committing them to see whether the changes have been applied by LEVIATHAN as intended.

The actual heuristics algorithm and, with it, the question which decisions to make in critical corner cases, is currently still work in progress. There are lots of different possibilities how to match context lines that are unchanged or that changed only to a certain degree, and how to match altered variant blocks back to source configuration blocks to apply the write-back operation. In order to offer an algorithm that proves to be valuable in practice, we are preparing an analysis of typical changesets in a couple of software product lines to be able to make an informed decision about the heuristics parameters.

4.2 Marker-Based Write-Back

If the developer's changes on a mounted view have to be a hundred percent unambiguous, LEVIATHAN offers an additional, optional mechanism called markers. Markers are language-dependent comment lines that delineate former configuration blocks in the variant views. If LEVIATHAN is configured to generate marker lines at mount time, those lines are generated by the serializer component when presenting the virtual file contents to the applications (see Figure 2). By leaving the marker lines in place and only editing the lines between them, the LEVIATHAN user can fully convey what he intends to change and how.

Upon saving the changes, LEVIATHAN's write-back parsing component parses the altered virtual file for the marker lines and maps the variant blocks in between to configuration blocks in the configurable code base (see Figure 2). The merge component then updates the file block structure in LEVIATHAN's cache for future read accesses; another serializer component writes back the altered file together with all of its original preprocessor directives to the base filesystem.

With its marker mechanism, LEVIATHAN's write-back support is completely unambiguous at the cost of slightly increased clutter through the introduced marker lines. However, the mounted variant will still be a lot more maintainable than the original configurable code base, since *non-active* configuration blocks and the corresponding directives are not visible in it; only *active* blocks and their markers are displayed in the view.

Optionally, editors can implement light-weight plug-ins to interpret LEVIATHAN's markers and highlight the information in an editor-specific way, for example using vim's folds or CIDE's colors. This means that those tools can be used *complementary* to LEVIATHAN. In that way, those tools can be seen as the independent view parts of a model–view–controller architecture; the actual preprocessing part is provided by LEVIATHAN.

5. DISCUSSION

In this section, we discuss how LEVIATHAN compares to existing tooling and how integration with those tools may be achieved. Additionally, we discuss current limitations of our approach and how to overcome them.

5.1 Using and Integrating Other Tools

Some of the use cases identified in Section 2 can also be addressed with existing tooling. However, our approach is the only one that is toolchain-independent and, thus, allows to work on a variant with arbitrary file-based tools. In the following, we will address each of the four use cases identified in Section 2.

When a tool works on a variant file read-only (e.g., for *WCET analysis*), a separate preprocessor tool could be applied to the code base before analysis. Integrating the preprocessor with the filesystem may be more convenient than manually executing an external preprocessor, but basically both perform the same task equally well. Variability-aware *code reasoning* up to now has required dedicated viewers and editors such as CIDE [6] or C-CLR [13]. Our solution is generic and can be used both with the developer's favorite open source editor as well as prescribed fixed editors in industrial settings. In case of *feature-local refactorings*, some refactorings might be done with semantic patch tools such as

Coccinelle [12]. However, Coccinelle detects semantic contexts based on matching normalized source code strings only. As the expressions are not evaluated, more complex Boolean conditions might be matched erroneously, resulting in patching the wrong set of code blocks. Furthermore, the patch transformations must be formulated in the Coccinelle language, whereas, with the LEVIATHAN filesystem, arbitrary tools, such as sed, Perl, or source code transformation languages such as TXL [3] may be used. *Maintenance changes* as well can be performed on a specific variant and be written back to the source code base using the developer's editor of choice.

Although LEVIATHAN's toolchain independence allows developers to use arbitrary editors and IDEs to work on mounted variants, even in scenarios where a developer employs variability-*enabled* editors such as CIDE [6] or FeatureMapper [5], LEVIATHAN may come in handy. As both have their own means for internally dealing with variability, LEVIATHAN could be used to transparently supply them with the variability file format they require, while the actual source code variability is managed with a preprocessor such as CPP. This means that those tools can be used *complementary* to LEVIATHAN. In that way, those tools can be seen as the independent view parts of a model–view–controller architecture; the actual preprocessing part is provided by LEVIATHAN. For this purpose, the expression evaluator (see also Figure 2) would be dispensable, as these tools do not work on variant files, but on unconfigured code bases. Furthermore, to actually integrate such tools, our serialization and parsing mechanisms need to be adapted accordingly in order to be able to write and read the variability file formats of tools such as CIDE and FeatureMapper.

5.2 Limitations of the Approach

One current limitation of our approach is that it does not support changing the *structure* of conditional blocks in a mounted view. This means that it is not possible to add, remove, or change the inclusion condition of such a block when working on the mounted view. This limitation is unproblematic for such use cases as *feature-local refactorings* and incremental *maintenance changes* (as described in Section 2), which do not affect the conditional structure. If, however, changes to the conditional structure are necessary, those changes can be performed directly on the configurable code base. By means of its internal notification mechanism (see also Section 3), LEVIATHAN will be able to update all of its mounted views where needed.

As mentioned before, LEVIATHAN's CPP component only evaluates the subset of CPP constructs used for conditional compilation such as #if, #ifdef, and #ifndef; it leaves out #include or #define statements. As a drawback we currently cannot definitely evaluate expressions containing CPP *macros*. However, only 2 of the 27,569 conditional expressions used for feature-based configuration in Linux[3] call a macro function (e.g., `#if LINUX_VERSION_CODE >= KERNEL_VERSION(2,2,0)` to query the kernel version). We deal with such cases by simply setting such expressions to undecided, which results in the inclusion of the corresponding block including its CPP annotations into the preprocessed file.

One very general concern about any tool that provides views on configurable code bases (such as LEVIATHAN) is the effect of a local feature change that was performed in a view on *other features* that are not visible in the current view. Consider, for instance, renaming a variable that is also used in a hidden feature block; this refactoring will make any variant that uses that feature stop from even compiling. If such problems are to be avoided, either the write-back results can be double-checked in the configurable code base, or the change can be performed directly in the code base itself, thereby effectively avoiding LEVIATHAN's advantage of taming #ifdef clutter. This has to be decided on a case-by-case basis, and some of the analyzed use cases (see Section 2) will be more susceptible to that problem than others.

[3]Each preprocessor variable used for configuration starts with the prefix `CONFIG_`.

6. CONCLUSION AND FUTURE WORK

We have shown a way to deal with the complexity of preprocessor-configured software—by using views as provided by our LEVIATHAN filesystem. Our approach improves on those based on special IDEs since it enables the use of arbitrary toolchains that work directly on files. This is crucial both in industry settings with fixed toolchains as well as in open-source projects, where very heterogeneous tools and development environments are used. Although some tools may in fact *be* #ifdef-aware, LEVIATHAN *modularizes* preprocessor functionality by implementing it on the filesystem level, providing true separation of concerns.

We still need to fully evaluate our LEVIATHAN approach and especially its write-back engine and approaches to be able to exactly state its benefits and disadvantages; the evaluation targets will be Linux and eCos, as well as a department-internal operating system product line that is used in classes. In future work, we additionally want to tackle read and write support for code bases that implement optional features with patch sets, which are, for example, prevalent for freshly implemented and experimental features in Linux. Furthermore, we are working on a formalization of our two write-back approaches to be able to grasp their respective advantages and boundaries—and, therefore, their applicability to different use cases.

7. REFERENCES

[1] 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 2010-08-23.

[2] Wave V2.0: Wave C++ preprocessor library. `http://www.boost.org/doc/libs/1_43_0/libs/wave/index.html`, visited 2010-07-29.

[3] J. Cordy. The TXL source transformation language. *Science of Computer Programming*, 61(3):190–210, Aug. 2006.

[4] FUSE: Filesystem in userspace. `http://fuse.sourceforge.net/`, visited 2010-07-29.

[5] F. Heidenreich, J. Kopcsek, and C. Wende. FeatureMapper: Mapping features to models. In *Proceedings of the 30th International Conference on Software Engineering (ICSE '08)*, pages 943–944, New York, NY, USA, 2008. ACM Press.

[6] C. Kästner, S. Apel, and M. Kuhlemann. Granularity in software product lines. In *Proceedings of the 30th*

International Conference on Software Engineering (ICSE '08), pages 311–320, New York, NY, USA, 2008. ACM Press.

[7] C. W. Krueger. BigLever software Gears and the 3-tiered SPL methodology. In *Companion to the 22nd ACM SIGPLAN Conference on Object-Oriented Programming Systems and Applications (OOPSLA '07)*, pages 844–845, New York, NY, USA, 2007. ACM.

[8] J. Liebig, S. Apel, C. Lengauer, C. Kästner, and M. Schulze. An analysis of the variability in forty preprocessor-based software product lines. In *Proceedings of the 32nd International Conference on Software Engineering (ICSE '10)*, New York, NY, USA, 2010. ACM Press.

[9] D. Lohmann, F. Scheler, R. Tartler, O. Spinczyk, and W. Schröder-Preikschat. A quantitative analysis of aspects in the eCos kernel. In *Proceedings of the ACM SIGOPS/EuroSys European Conference on Computer Systems 2006 (EuroSys '06)*, pages 191–204, New York, NY, USA, Apr. 2006. ACM Press.

[10] GNU M4 – GNU Project – Free Software Foundation (FSF). `http://www.gnu.org/software/m4/`, visited 2010-07-29.

[11] A. Massa. *Embedded Software Development with eCos*. New Riders, 2002.

[12] Y. Padioleau, J. L. Lawall, G. Muller, and R. R. Hansen. Documenting and automating collateral evolutions in Linux device drivers. In *Proceedings of the ACM SIGOPS/EuroSys European Conference on Computer Systems 2008 (EuroSys '08)*, Glasgow, Scotland, Mar. 2008.

[13] N. Singh, C. Gibbs, and Y. Coady. C-CLR: A tool for navigating highly configurable system software. In *Proceedings of the 6th AOSD Workshop on Aspects, Components, and Patterns for Infrastructure Software (AOSD-ACP4IS '07)*, pages 1–6, New York, NY, USA, 2007. ACM Press.

[14] H. Spencer and G. Collyer. #ifdef considered harmful, or portability experience with C News. In *Proceedings of the 1992 USENIX Annual Technical Conference*, Berkeley, CA, USA, June 1992. USENIX Association.

TypeChef: Toward Type Checking #ifdef Variability in C

Andy Kenner
Metop Research GmbH
Magdeburg, Germany
andy.kenner@metop.de

Christian Kästner
Philipps University Marburg
Marburg, Germany
kaestner@informatik.uni-marburg.de

Steffen Haase,
Thomas Leich
Metop Research GmbH
Magdeburg, Germany
haase/leich@metop.de

ABSTRACT

Software product lines have gained momentum as an approach to generate many variants of a program, each tailored to a specific use case, from a common code base. However, the implementation of product lines raises new challenges, as potentially millions of program variants are developed in parallel. In prior work, we and others have developed product-line–aware type systems to detect type errors in a product line, without generating all variants. With *TypeChef*, we build a similar type checker for product lines written in C that implements variability with *#ifdef* directives of the C preprocessor. However, a product-line–aware type system for C is more difficult than expected due to several peculiarities of the preprocessor, including lexical macros and unrestricted use of *#ifdef* directives. In this paper, we describe the problems faced and our progress to solve them with *TypeChef*. Although *TypeChef* is still under development and cannot yet process arbitrary C code, we demonstrate its capabilities so far with a case study: By type checking the open-source web server *Boa* with potentially 2^{110} variants, we found type errors in several variants.

Categories and Subject Descriptors

D.3.3 [**Programming Languages**]: Language Constructs and Features; D.3.4 [**Programming Languages**]: Processors—*Preprocessors*; D.2.13 [**Software Engineering**]: Reusable Software

General Terms

Languages, Reliability, Theory

Keywords

Type system, conditional compilation, C, cpp, #ifdef, partial preprocessor, disciplined annotations

1. INTRODUCTION

Software product line engineering is an efficient means to implement variable software. By selecting from a set of features, a developer can generate different program variants from a common product-line implementation. However, variability comes at a price of increased complexity. Instead of developing and testing a single variant, developers deal with potentially millions of variants in parallel. Already with a few features, we quickly reach a point at which it is no longer possible to compile and run every possible variant in isolation, due to the vast number of variants (up to 2^n variants for n features).

To address this problem, researchers have developed mechanisms that check certain criteria for the entire product line, instead of checking each variant in isolation. This ranges from simple guarantees of syntactic correctness [22], to dead-code detection [34, 35], to type checks and similar referential-consistency checks [1, 2, 8, 16, 21, 36], and to behavioral checks using, among others, model checking [7, 14, 24, 31]. Usually, the idea is to analyze source code before variant generation when it still includes its variability mechanisms; the approaches check implemented variability against the variability model, which describes all valid feature combinations.

Especially, product-line–aware type checking (or reference checking, or safe composition) has shown to scale [8, 20, 36]; type checking an entire software product line with millions of variants is usually as fast as checking a handful of variants in isolation. Product-line–aware type checking has been explored for different variability implementations, most prominently for AHEAD-style feature modules and class refinements [1, 36] and for annotation-based implementations (typically using some form of conditional compilation, also known as negative variability) [2, 8, 16, 21]. However, corresponding type systems were usually targeted at dialects of Java (or Featherweight Java) which limited their applicability to industrial software product lines.

A typical setting to implement variability in industrial software product lines is to use C as programming language [17] and use conditional-compilation directives (*#ifdef*, *#if*, *#elif*, *#else*, *#endif*) of the C preprocessor *cpp* to implement variability (despite broad criticism on the C preprocessor, which is out of scope here). Although we are unaware of any statistics or surveys on industrial product-line implementation, our personal communication with tool providers and developers indicates that actually a majority of industrial software product lines are implemented with the C preprocessor. For example, HP's product line Owen for printer firmware with over 2000 features implements variability entirely with *cpp* [29, 32]; so do many open source programs [25], of which the Linux kernel with over 8000 features is probably the most prominent example [25, 33, 35]. Three more examples of industrial product lines presented at last year's Software Product Line Conference that implement variability at least partially with preprocessors are Danfoss' product line of frequency converters [18], Wikon's product line of remote control systems [30], and NASA's product line of flight control systems [11].

Our overall goal is to type check an entire product line written in C and *cpp* with all its variants. To this end, we construct a product-line–aware type checker called TypeChef (type checking *#ifdef*s). Unfortunately, the transfer from a confined research setting in the Java environment to industrial C code turned out much harder than expected. Even though C's type system is rather simple, the C preprocessor *cpp* makes analysis difficult. In a nutshell, *cpp* works on token level; it can be (and is) used at fine granularity and in patterns that are very hard to understand by analysis tools. Additionally, file inclusion and lexical macro substitution (which were not present in previous Java settings) interfere with our analysis, especially when macros are conditionally defined or have alternative expansions. Finally, also presence conditions for code are more complex; *cpp* does not only allow propositional formulas after *#if* directives, but also integer constants (which may be defined, redefined, or undefined during preprocessing) and various operations on them.

In this paper, we describe the problems of type checking a product line implemented with C and *cpp* and describe our solution with TypeChef so far. Specifically, we designed a partial preprocessor to tame *cpp* directives to a disciplined level; and we designed and implemented a type checker that understands *#ifdef* directives and checks them in a way similar to previous product-line–aware type systems. TypeChef is work in progress; we cannot parse arbitrary C code, yet; we do not support alternatives and some manual code preparation is still necessary. Nevertheless, we have already applied *TypeChef* to a small open-source implementation of the Boa web server with 110 features and found several inconsistencies.

In summary, we make the following contributions: (1) We outline the difficulties of product-line–aware type checking for C code. (2) We present and implement an initial solution with TypeChef. (3) In that context, we propose the concept of a partial preprocessor to handle file inclusion and macro substitution. (4) We demonstrate how TypeChef can detect inconsistencies in a small case study.

2. PRODUCT-LINE–AWARE TYPE CHECKING

Let's start by revisiting the basic idea behind product-line–aware type systems, without considering the particularities of *cpp*, yet.

Consider the trivial code fragment in Figure 1. It does nothing more than output a single line of text. However, which text this is depends on preprocessor flags (or features in a product-line context). Lines 4 and 7 are only compiled if the corresponding features are selected; otherwise *cpp* removes the code before compilation. To describe when a code fragment is included, we speak of a *presence condition* $\mathtt{pc}$; a code fragment is only included when its presence condition evaluates to true for the given feature selection. Line 4 has the presence condition $\mathtt{pc}(\mathit{line4}) = \mathit{WORLD}$, i.e., it is only included when feature WORLD is selected.[1] This small program, which we can consider as product line, has two features (WORLD and BYE) and can generate four possible variants (with neither feature, with both features, or with either feature). Only two of these four variants will compile though. The compiler will issue an error for the second definition of variable *msg ("Line 7: redefinition of msg")* when both features are selected and will issue an error about a dangling reference *("Line 11: msg undeclared")* when neither feature is selected.

With a product-line–aware type system, we want to guarantee that all potential variants of a product line are well-typed, without

```
 1 #include <stdio.h>
 2
 3 #ifdef WORLD
 4 char * msg = "Hello␣World\n";
 5 #endif
 6 #ifdef BYE
 7 char * msg = "Bye␣bye!\n";
 8 #endif
 9
10 main() {
11   printf(msg);
12 }
```

Figure 1: Example C program

generating all variants. That is, we want to check types before running the preprocessor with a specific feature combination but still guarantee that all variants compile after generation.

In a nutshell, we resolve references and compare annotations in the original code, typically based on an underlying abstract-syntax-tree representation. In Figure 1, *printf* references a function that is declared in the included *stdio.h* file and *msg* references the declarations in Line 4 or 7. Based on these reference pairs, we compare the annotated features. The function call *printf* in Line 11 is not annotated by a feature, neither is the function declaration in *stdio.h*; hence, both call and declaration are included in all variants and do not cause type errors. However, variable *msg* is only declared when feature WORLD or feature BYE is included, but referenced in all variants, so we can predict that some variants will not compile. Similarly, we can identify that there are variants in which both variable declarations are included at the same time and issue an error.

In most cases, checking annotations regarding all possible feature combination is too strict though. Typically, domain experts restrict possible feature combinations in a product line, for example, by specifying that either feature WORLD or BYE has to be selected in every variant. In product-line engineering, it is best practice to document such domain knowledge in *variability models*. A variability model describes the *intended* variability of the program. Typical forms of variability models are *feature models* and their graphical representation as *feature diagrams* [19], but some projects, such as the Linux kernel, have their own variability-modeling languages [35].

A product-line–aware type system can use a variability model as input and type check only variants allowed by the variability model, instead of all feature combinations. Mathematically, checking only allowed variants is expressed as $\mathit{VM} \rightarrow (\mathtt{pc}(\mathit{caller}) \rightarrow \mathtt{pc}(\mathit{target}))$; that is, the presence condition of the caller must imply the presence condition of the target in all variants allowed by the variability model VM. Similarly, we can check for redefinitions of variables or functions with $\mathit{VM} \rightarrow \neg(\mathtt{pc}(\mathit{def1}) \wedge \mathtt{pc}(\mathit{def2}))$. If the formula is not a tautology (determinable by a SAT solver or other solvers), we issue an error message and can provide an example of a feature selection that causes a type error. For reasoning in a type system, most kinds of variability models can be translated directly into logics [4, 36], and reasoning about them is tractable for even very large models [27]. Our experience shows that the time spent by SAT solvers to determine tautologies is negligible compared to the remaining lookup processes [20].

Given a feature model $\mathit{VM} = (\mathit{WORLD} \vee \mathit{BYE}) \wedge \neg(\mathit{WORLD} \wedge \mathit{BYE})$ that defines that exactly one feature must be selected in all variants, we can statically guarantee that the code from Figure 1 is well-typed in all variants: the check regarding *printf* is trivially a tautology ($\mathit{VM} \rightarrow \mathtt{true} \rightarrow \mathtt{true}$), the reference check regarding

[1]Deriving presence condition from *#ifdef*, *#elif*, and *#else* directives (including nesting) is straightforward; for a formal definition see [34]. A code fragment that is not nested in *#ifdef* directives has the presence condition $\mathtt{true}$, i.e., it is included in all variants.

msg is a tautology ($VM \rightarrow \texttt{true} \rightarrow (WORLD \vee BYE)$), and also the condition to prevent redefinition is a tautology ($VM \rightarrow \neg(WORLD \wedge BYE)$).

Checked Properties. Our aim is to find type errors. That is, we want to find the same errors for the entire software product line that a compiler would find for a specific variant. We neither address dynamic properties nor further static properties beyond the type system, such as single assignment. That is, we ensure that each variant compiles, but not that it has meaningful runtime semantics. Adopting static analysis and behavioral checks to software product lines are interesting but separate research challenges [7, 14, 24, 31].

The type system of C is considered as weak, because of implicit type conversion, and unsafe, mainly because of casts between pointers. Chandra and Reps [6] summarize "In C, a pointer of a given type can be *cast* into any other pointer type. Because of this, a programmer can interpret any region of memory to be of any type. Traditional type checking for C cannot enforce that such reinterpretation of memory is done in a meaningful way, because the C standard allows arbitrary type conversions between pointer types. For this reason, C compilers and tools such as *lint* do not provide any warnings against potential runtime errors arising from the use of casts."

Still, there are many kinds of errors that the C type system detects, including dangling variable references (as in Figure 1), dangling function calls, function calls with an incorrect number of parameters, redefinitions of functions and variables, references to undefined types, and type mismatch for assignments and function arguments [17]. Our long-term goal is to cover the entire type system of C as specified in the standard [17]. Nevertheless, we start by checking references (to variables, functions, type declarations), because they are most problematic in a product-line setting, in our experience.

3. PARSING PRE-CPP CODE IS HARD

The main challenge in type checking C code is to parse C code that still contains *cpp* directives (pre-*cpp* code) into a representation that allows us to look up presence conditions and references between elements. Already parsing preprocessed C code is difficult in practice [5], but parsing pre-*cpp* code is a difficult challenge, a challenge already faced by many refactoring and code-analysis tools [3, 9, 10, 13, 26, 28, 37]. Solution strategies either use heuristics [13, 28]—which is not suitable for type checking, since we want to give guarantees—or parse only a subset of possible input programs [3, 26].

A common subset strategy is to build a parser that understands C code with preprocessor directives at certain locations only. For example, *#ifdef* directives may only wrap entire functions or statements, but not arbitrary tokens. We call such restricted use of *#ifdef* directives *disciplined annotations*. Given disciplined annotations, we can create an abstract syntax tree and assign presence conditions to subtrees. Unfortunately, enforcing disciplined annotations may be realistic when writing new code; but an earlier large-scale analysis of 40 cpp-based product lines with a total of 30 million lines of code [20] has shown that on average 11 % of all *#ifdef* directives are *not* in a disciplined form. Consequently, without manual preparation or further tool support, hardly any file can be parsed with this approach.

Additional difficulties come from macro substitution and file inclusion. To parse C code (even when all conditional compilation directives are disciplined), *#include* directives and macros must be expanded. That is, although we want to parse pre-*cpp* code, we need to handle file inclusion and macro substitution during parsing

```
#define P(msg) \
    printf(msg);

main() {
  P("Hello\n")
  P("World\n")
}
```

(a) Macro-defined syntactic structure

```
#ifdef BIGINT
#define SIZE 64
#else
#define SIZE 32
#endif

allocate(SIZE);
```

(b) Alternative macro expansions

```
#ifdef SIZE==64
#define BIGINT
#endif

#ifdef BIGINT
//...
#endif
```

(c) Conditionally defined features

Figure 2: Difficulties in parsing C code

nevertheless. For example, we can only check the reference of the function call *printf* in Figure 1, if we include and parse *stdio.h* first (and recursively the files it includes). To parse the statements in Figure 2a, we need to expand the macro first, which inserts the semicolon necessary to parse the body as two statements. To make matters worse, a macro can have alternative expansions as shown in Figure 2b, and we might need definitions of macros (which may depend on other macros) in future presence conditions as illustrated in Figure 2c.

Parsing pre-*cpp* code is the main challenge for type checking C code, whereas detecting references on an abstract syntax tree and checking presence conditions against a feature model is a straightforward adaptation. Undisciplined annotations, macro substitution, and include directives were all not problems in prior approaches based on Java and its restricted preprocessors or language extensions. In the next section, we describe how we tackle these problems and present a first solution with our tool *TypeChef*.

4. AN OVERVIEW OF TYPECHEF

TypeChef addresses the problem of analyzing pre-*cpp* code in multiple steps—partial preprocessor, expansion to disciplined annotations, parsing, reference analysis, and solving—as illustrated in Figure 3. We discuss each step and its challenges and solutions in isolation.

4.1 Partial Preprocessor

First, we are interested in pre-*cpp* code because of its variability. Nevertheless, we have to expand macros and file inclusions to be able to parse the source code at all. To this end, we contribute a *partial preprocessor*: We pursue the strategy to process macros and file inclusion without affecting variability of conditional compilation constructs. In the example of Figure 3, we recursively include all code from *stdio.h* (for brevity we show only the declaration of method *printf*) and replace all occurrences of the macro *T* by its expansion *char* *. Note that the *#ifdef* directives are not changed.

Technically, we currently use a simple hack to implement the partial preprocessor. With a script, we comment out all *#ifdef* directives (except include guards, see below) as illustrated in Figure 4, then run the original preprocessor (which now sees only *#include* and *#define* directives and processes them as usual), and finally remove the comments to restore the *#ifdef* directives. Regarding file inclusion, the preprocessor already provides *#line* directives to maintain information where code came from; this is important to display error messages at the correct location later on. Regarding macro expansion, we do not store information about expansion, yet.

Include guards deserve special attention. An include guard is a standard pattern in C to prevent multiple or recursive inclusions of a file; it uses the same *#ifdef* or *#ifndef* directives as feature code, but follows the pattern illustrated in Figure 5. The partial preprocessor must process include guards, because otherwise indefinite loops can

```
#include <stdio.h>
#define T char *
main(){
  T msg =
#ifdef WORLD
      "Hello_World\n";
#else
      "Bye_Bye!\n";
#endif
  printf(msg);
}
```

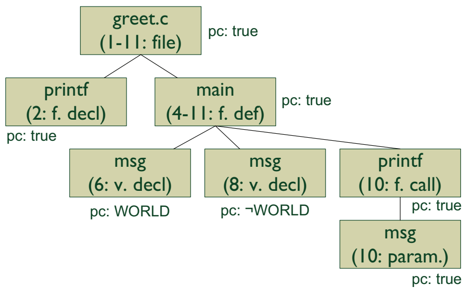
$\boxed{\text{(1) Partial Preprocessor} \Downarrow}$

```
...
int printf(const char *, ...);
...
main(){
  char * msg =
#ifdef WORLD
      "Hello_World\n";
#else
      "Bye_Bye!\n";
#endif
  printf(msg);
}
```

$\boxed{\text{(2) Expansion to Disciplined Annotations} \Downarrow}$

```
...
int printf(const char *, ...);
...
main(){
#ifdef WORLD
  char * msg = "Hello_World\n";
#else
  char * msg = "Bye_Bye!\n";
#endif
  printf(msg);
}
```

$\boxed{\text{(3) Parsing} \Downarrow}$



$\boxed{\text{(4) Reference Analysis} \Downarrow}$

$$VM \rightarrow (\mathit{true} \rightarrow \mathit{true})$$

$$VM \rightarrow (\mathit{true} \rightarrow (WORLD \vee \neg WORLD))$$

$$VM \rightarrow \neg(WORLD \wedge \neg WORLD)$$

$\boxed{\text{(5) Solving} \Downarrow}$

"all variants are well-typed"

Figure 3: TypeChef steps

```
#include <stdio.h>
#define T char *
main(){
  T msg =
//#ifdef WORLD
      "Hello_World\n";
//#else
      "Bye_Bye!\n";
//#endif
  printf(msg);
}
```

Figure 4: Intermediate step of the partial preprocessor: Commenting out conditional compilation directions

```
#ifndef _FLAG
#define _FLAG
...
#endif
```

Figure 5: Include-guard pattern

occur. Hence, we do not comment out preprocessor directives that belong to include guards. Fortunately, we do not need to consider include guards as variability in our type system; it is acceptable to not have them in presence conditions. To distinguish between include guards and *#ifdef* directives that implement variability, we currently use a pattern matching mechanism: *TypeChef* recognizes *#ifndef* and *#define* directives at the beginning of a file and *#endif* at the end of a file as include guard. Alternatively, we could rely on naming conventions, such as "flags for include guards start with an underscore", which are used in most projects anyway.

This simple partial preprocessor, based on comments, works reasonably well. However, it has two limitations, which we currently address in ongoing work. First, and most importantly, we cannot support alternative macro definitions as illustrated in Figure 2b. Second, we cannot use previously defined macros in the condition of an *#if* directive as shown in Figure 2c. In ongoing work, we are developing a more sophisticated partial preprocessor, which can handle alternative macro definitions (by introducing additional *#ifdef* directives at expansion) and conditionally defined feature flags (roughly based on prior work on symbolic execution of *cpp* directives [15,23]).

4.2 Expansion to Disciplined Annotations

In a second step, we enforce disciplined annotations. As disciplined annotations, *TypeChef* currently allows *#ifdef* directives that wrap one or more entire top-level declarations and definitions (i.e., declarations or definitions of function, structures, unions, and global variables) and directives that wrap one or more statements inside a function, or fields inside a structure or union. In contrast, *TypeChef* considers conditional compilation directives at finer granularity or around partial elements as undisciplined.

In general it is always possible to expand undisciplined annotations to disciplined ones (not all of these expansions are necessarily parseable or well-typed, of course). In the worst case, we can use a brute-force mechanism which replicates the entire code for every possible feature combination. To prevent the exponential complexity, expansions at finer granularity are useful. For example, in Figure 3, we replicate the statement and have two alternative statements instead. In many cases, it might also be possible to manually rewrite to code into a disciplined form, often by introducing additional variables.

```
 1 compilation_unit: external_declaration*;
 2 external_declaration:
 3   function_def |
 4   variable_def |
 5   '#if' cppexp '\n' external_declaration '\n' cppthenfunc;
 6 cppthenfunc:
 7   '#endif' '\n' |
 8   '#else' '\n' external_declaration '\n' '#endif' '\n' |
 9   '#elif' cppexp '\n' external_declaration '\n' cppthenfunc;
10 function_def ...
```

Figure 6: Extended C grammar

Name	Type	Scope	Presence Condition
`printf`	`char *` $\rightarrow$ `int`	0	$true$
`msg`	`char *`	0	$WORLD$
`msg`	`char *`	0	$\neg WORLD$

Figure 7: Extended symbol table for the example from Fig. 3

Currently, *TypeChef* does not yet automate this step, but a developer has to manually expand undisciplined annotations. In related work, Garridio has implemented such expansion for refactoring C code [12]; we plan a similar tool to automate the task.

4.3 Parsing

Once we have included all files, substituted all macros, and enforced disciplined annotations, the remaining parsing is straightforward. We take a standard C grammar and extend it with productions of *#ifdef* directives as illustrated in Figure 6 (Lines 7–11 are added to detect *#ifdef* directives around top-level declarations). From such grammar, we generate a parser for *TypeChef*, which produces an abstract syntax tree. Parsed *#ifdef* directives are either part of this tree or can be reduced to presence conditions that are annotated at every structural element as shown in Figure 3. For *TypeChef*, we have implemented such parser with the parser generator ANTLR, based on an existing GNU C grammar.

4.4 References Analysis

Based on the abstract syntax tree with presence conditions, TypeChef now looks up references that should be checked. As a result of this step, TypeChef creates a set of formulas (one for each reference or one conjunct formula for all references) that we can later feed into a solver.

Reference lookup in C is mostly straightforward by iterating once over the abstract syntax tree.[2] A simple symbol table, as in Figure 7, is sufficient to store all declared types, variables, and functions, and their respective type, scope,[3] and presence condition. Whenever, we reach a declaration, we add a corresponding entry in the symbol table; in case already an entry with the same name is present (or even multiple), we produce a formula in the form $VM \rightarrow \bigwedge_i \neg(pc(newDecl) \wedge pc(prevDecl_i))$ to check that all declarations are mutually exclusive. When we find a function call (or variable access or reference to a type), we look up the function's (variable's, type's) name in the symbol table and retrieve the corresponding presence condition(s). We then produce a corresponding formula $VM \rightarrow (pc(caller) \rightarrow \bigvee_i pc(decl_i))$.

[2]Technically, we implemented two iterations, which, however, could be merged.

[3]A scope is necessary for variables to distinguish between variables defined globally or in a function. The distinction is not relevant in our small examples in this paper.

We check references to fields in structures and unions in a similar way; the only difference is that we need to look up the type of a local variable first. That is, right now, we can guarantee that compilation will not fail due to dangling function invocations, and dangling references to variables, fields, or types. In ongoing work, we additionally add checks to ensure consistency between function declarations and functions definitions, and to ensure matching types for assignments, function arguments, and so on; so far, we check only simple references. Also matching signatures in different object files, as checked by the linker, will be addressed.

4.5 Solving

Finally, we need to solve the formulas produced during reference analysis. Throughout this paper, we used propositional formulas for presence conditions and variability models. Actually, the C preprocessor supports more than that: It additionally supports numeric constants and various operations, such as sum, comparison, and bitwise shifting [17]. Therefore, *TypeChef* encodes presence conditions and feature models as constraint satisfaction problem as described by Benavides et al. [4], instead of using a propositional formula. Technically, *TypeChef* uses the constraint-satisfaction-problem solver *Choco*[4] to check for tautologies. We check the formula for each reference in isolation (instead of building one big formula), so that we can trace an error directly to the reference which causes it.

In case a formula is not a tautology, *Choco* finds a counter example representing a specific variant which will not compile. We can present the counter example to the user for further debugging. We can produce an error message that mimics the style of a C compiler (file, line, reason) and that additional provides information about problematic variants.

5. CASE STUDY: BOA WEBSERVER

We implemented *TypeChef* as outlined above. As discussed, *TypeChef* is still work in progress, and there are still significant limitations which prevent applying it to a large-scale industrial C project. Especially, the manual expansion of undisciplined annotations is a severe restriction. Still, we want to demonstrate *TypeChef* at this stage with a (favorable) case study.

As subject of our case study, we selected the open-source web server Boa, version 0.94.13.[5] Boa is a lightweight, single-threaded, and fast implementation of a web server, used mostly in embedded systems and for fast delivery of static content (e.g., `slashdot.org` uses it to deliver image files). It is written in C (6 200 LOC; 38 files) and contains some variability implemented with *cpp*'s *#ifdef* directives. Together, there are 110 different *#ifdef* flags. Some of these flags deal with low-level portability issues, but several can be considered as features in the sense of a product line, for example GUNZIP to support packed HTML files, USE_LOCALTIME to switch between local time and GMT, INET6 to switch between IPv4 and IPv6, three alternative hashing algorithms, and several debug options (logging levels, extra supervision for hash tables, and others). Unfortunately, features and their dependencies are not documented. In theory, there are up to 2^{110} variants of Boa. Even if we consider only some *#ifdef* flags that correspond to end-user variability in a product-line sense (see examples above), we estimate about a thousand possible variants. Hence, generating and checking all variants in isolation does not scale.

We selected Boa because of its manageable size and because almost all of its *#ifdef* directives are in a disciplined form already. Af-

[4]`http://www.emn.fr/z-info/choco-solver/`

[5]`http://www.boa.org/`

```
#ifdef YYPARSE_PARAM
int yyparse (void
      *YYPARSE_PARAM)
#else
int yyparse (void)
#endif
{
  //method body
}
```

(a) Original undisciplined implementation

```
#ifdef YYPARSE_PARAM
int yyparse (void
      *YYPARSE_PARAM)
{
  //method body
}
#else
int yyparse (void)
{
  //method body
}
#endif
```

(b) Expanded disciplined implementation

Figure 8: Alternative method signatures in Boa

ter applying the partial preprocessor, we only needed to expand eight undisciplined annotations, such as the alternative method signatures in Boa's internal (generated) parser shown in Figure 8. Furthermore, neither alternative macros nor conditional feature definitions (cf. Sec. 4.1) cause serious complications in Boa. Boa is a favorable case study that is not significantly affected by the limitations of our current implementation. Nevertheless it is valuable to demonstrate feasibility of our approach and to encourage further improvements toward accepting more and larger C implementations.

With reference analysis, *TypeChef* detects 38 671 references within the entire implementation of Boa (including references within the included header files). These are 2 008 function calls, 7 250 references to variables, 21 934 references to types, and 7 479 references to fields of structures or unions. Of the 38 671 references, 35 478 (92 %) are obviously correct because the target code fragment is not wrapped by *#ifdef* directives or because both elements have the same presence condition. This left us with 3 193 references, which we handed to the solver. Of these, 2 171 (68 %) were tautologies, the remaining 1 022 references are potentially indicators of errors. Additionally, there were 138 potential references for which we did not find a target, which indicate dead or unmaintained code (or incorrect header files in our environment).

We have to be careful with interpreting the solver's results though. To the best of our knowledge, Boa does not have a variability model, neither explicitly nor implicit in some developer documentation. The build environment (*configure* script) does not help either. Nevertheless, domain knowledge that might have been obvious to the original developers might dictate certain dependencies between features, which we were not aware of. Even a single missing dependency in the feature model can lead to many error reports. Finally, there are some false positives caused by limitations of *TypeChef's* current implementation. Hence, we manually inspected the reported errors.

With manual inspection, we could confirm a small number of bugs or undocumented dependencies (we are not familiar enough with the source code to make that judgment). Here, we show two of them with a small code excerpt. First, as illustrated in Figure 9, the flag DEBUG must never be included in any variant, otherwise there will be a dangling reference to *h* in *mmap_cache.c*. Second, as illustrated in Figure 10, the flag HAVE_SYS_FCNTL_H must be included in all variants, otherwise there will be a compilation error due to unknown types in included headers of *alphasort.c*.

TypeChef needs about one minute to check all variants (parsing, reference analysis, and solving), whereas compiling a single variant requires about four seconds on the same system. Solving all equations for the 3 193 non-trivial references takes six seconds with the solver *Choco*. Hence, already with 20 variants, *TypeChef* is faster

```
struct mmap_entry *find_mmap(int data_fd, struct stat *s)
{
    char *m;
    int i, start;
    ...
#ifdef DEBUG
    fprintf(stderr, "New␣mmap␣list␣entry␣%d␣(hash␣was␣%d)\n",
        i, h);
#endif
```

Figure 9: Detected bug or undocumented dependency in includes of file *mmap_cache.c*

```
#ifdef HAVE_SYS_FCNTL_H
...
  typedef __darwin_off_t off_t;
  typedef __darwin_pid_t pid_t;

...
#endif

int sendfile(int, int, off_t, off_t *, struct sf_hdtr *, int);
...
pid_t fork(void);
```

Figure 10: Detected bug or undocumented dependency in includes of file *alphasort.c*

than the brute-force strategy of compiling all variants in isolation, which is in line with prior experience in product-line–aware type systems for Java [20].

6. RELATED WORK

There have been many approaches to analyze pre-*cpp* code for various purposes. One driving factor were refactorings, which—compared to Java or Smalltalk—are very difficult to implement for pre-*cpp* C code. For example, Vittek used a brute-force-expansion mechanism as sketched in Section 4.2 [37] and Garridio developed a sophisticated mechanism to expand undisciplined annotations at fine granularity [12, 13]. With some heuristics, Garridio's tool was also able to deal with macro expansion and file inclusion, and it could propagate changes back to the original pre-*cpp* code. Similarly, Padioleau uses a sophisticated mechanism based on heuristics (including a significant amount of per-project heuristics) to parse pre-*cpp* code [28]. However, for type checking, we do not want to rely on heuristics; hence, we decided to use a simpler but more accurate mechanism of a partial preprocessor, which is sufficient for type analysis (but would not have been sufficient for refactoring because it cannot propagate changes back).

Additionally, there are several approaches that analyze *cpp* without looking at the underlying code. For example, Tartler et al. analyze C code (including the Linux kernel) for dead code, which cannot be included in any variant [35] and extract presence condition for every code fragment [34]. With a related goal, Hu et al. and Latendresse used symbolic execution to determine presence conditions for all code fragments, also for cases in which features are defined or undefined within the source code as in Figure 2c [15, 23]. Favre extracts the exact semantics of *cpp* for further analysis [10]. These approaches work on lines of arbitrary source code, whereas *TypeChef* looks in between preprocessor directives and analyzes the underlying C code regarding references.

Aversano et al. were the first to suggest to type check a C program including its *#ifdef* variability [2]. They primarily addressed alternative declarations with different types, in contrast to our focus on references. Their focus was low-level portability of C programs instead of variability in a product line setting, but the solutions and even proposed architectures are similar. In their work, they already proposed an extended symbol table as we used in Figure 7, but, unfortunately, this project was neither implemented nor continued.

The approach to parse pre-*cpp* C code by an extended grammar after manual preparation toward disciplined annotations is often credited to Baxter and Mehlicher [3], who also discussed their experience that 85 % of all *#ifdef* directives are disciplined and the remaining directives received manual attention. In addition, there are several suggestions to replace text-based preprocessors such as *cpp* with a more restricted preprocessor that know the underlying structure [22, 26, 38]. Such approaches restrict conditional compilation constructs to entire language fragments (as we do with disciplined annotations) and either abandon macros or propose syntax macros that are easier to handle. McCloskey and Brewer even provide a semi-automatic migration tool for their disciplined preprocessor *ASTEC* [26]. Unfortunately, we do not expect that we can force developers to switch to a different preprocessor (especially when huge amounts of legacy code are involved) or to manually change their implementations toward disciplined annotations. Therefore, in future work, *TypeChef* aims at preparing the source code automatically during analysis by partial preprocessing and (in the future) automatic expansion of undisciplined annotations.

Finally, in the context of more restricted languages (Java, Featherweight Java, UML), there have been many approaches to check for type errors, reference errors, and other kinds of errors in entire software product lines, e.g., [1, 7, 8, 16, 21, 31, 36]. Their details are beyond the scope of this paper, but the general idea, as outlined in Section 2, is similar in most of them. For a detailed discussion see [20].

7. CONCLUSION

The variability in software product lines provides many opportunities but also complicates development and testing, because a whole family of related variants is developed in parallel. Our goal is to detect implementation errors as early during product-line development and without compiling and testing every variant in isolation. With *TypeChef*, we transfer prior advances in type checking entire software product lines to industrial C code, in which variability is implemented with the C preprocessor *cpp*. Unfortunately, *cpp* has several characteristics that make analysis of unpreprocessed code very difficult. As we have described, to parse C code, we need to expand macros and file inclusion directives and we have to deal with preprocessor directives at every level of granularity and in many undisciplined forms that are difficult to handle.

TypeChef makes first steps toward making C code accessible for product-line–aware type checking. It combines several prior approaches to analyze pre-*cpp* code. With a partial preprocessor, we resolve macros and inclusion directives. With a specialized parser, we can subsequently parse disciplined *#ifdef* directives, analyze references and types within the source code, and detect errors with an off-the-shelf solver. *TypeChef* is work in progress, and in ongoing work, we address limitations, such as alternative macro expansions and undisciplined annotations. Nevertheless, we could already demonstrate the feasibility of *TypeChef* in a favorable case study, which is encouraging for further attempts to type check larger code bases of industrial C code. Our long-term goal is to soundly type check the entire Linux kernel with over 8 000 features and a well-specified variability model.

Acknowledgments. Käster's work is supported by the European Research Council (grant ScalPL #203099).

8. REFERENCES

[1] S. Apel, C. Kästner, Größlinger, and C. Lengauer. Type Safety for Feature-Oriented Product Lines. *Automated Software Engineering*, 17(3):251–300, 2010.

[2] L. Aversano, M. D. Penta, and I. D. Baxter. Handling Preprocessor-Conditioned Declarations. In *Proc. Int'l Workshop Source Code Analysis and Manipulation (SCAM)*, pages 83–92. 2002.

[3] I. Baxter and M. Mehlich. Preprocessor Conditional Removal by Simple Partial Evaluation. In *Proc. Working Conf. Reverse Engineering (WCRE)*, pages 281–290. 2001.

[4] D. Benavides, P. Trinidad, and A. Ruiz-Cortes. Automated Reasoning on Feature Models. In *Proc. Conf. Advanced Information Systems Engineering (CAiSE)*, pages 491–503. 2005.

[5] A. Bessey, K. Block, B. Chelf, A. Chou, B. Fulton, S. Hallem, C. Henri-Gros, A. Kamsky, S. McPeak, and D. Engler. A Few Billion Lines of Code Later: Using Static Analysis to Find Bugs in the Real World. *Commun. ACM*, 53(2):66–75, 2010.

[6] S. Chandra and T. Reps. Physical Type Checking for C. In *Proc. Workshop on Program Analysis for Software Tools and Engineering (PASTE)*, pages 66–75. 1999.

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

[8] K. Czarnecki and K. Pietroszek. Verifying Feature-Based Model Templates Against Well-Formedness OCL Constraints. In *Proc. Int'l Conf. Generative Programming and Component Engineering (GPCE)*, pages 211–220. 2006.

[9] J.-M. Favre. Understanding-In-The-Large. In *Proc. Int'l Workshop on Program Comprehension*, page 29. 1997.

[10] J.-M. Favre. CPP Denotational Semantics. In *Proc. Int'l Workshop Source Code Analysis and Manipulation (SCAM)*, pages 22–31. 2003.

[11] D. Ganesan, M. Lindvall, C. Ackermann, D. McComas, and M. Bartholomew. Verifying Architectural Design Rules of the Flight Software Product Line. In *Proc. Int'l Software Product Line Conference (SPLC)*, pages 161–170. 2009.

[12] A. Garrido. *Program Refactoring in the Presence of Preprocessor Directives*. PhD thesis, University of Illinois at Urbana-Champaign, 2005.

[13] A. Garrido and R. Johnson. Analyzing Multiple Configurations of a C Program. In *Proc. Int'l Conf. Software Maintenance (ICSM)*, pages 379–388. 2005.

[14] A. Gruler, M. Leucker, and K. Scheidemann. Modeling and Model Checking Software Product Lines. In *Proc. Int'l Conf. Formal Methods for Open Object-Based Distributed Systems (FMOODS)*, pages 113–131. 2008.

[15] Y. Hu, E. Merlo, M. Dagenais, and B. Laguë. C/C++ Conditional Compilation Analysis using Symbolic Execution. In *Proc. Int'l Conf. Software Maintenance (ICSM)*, pages 196–206. 2000.

[16] S. S. Huang, D. Zook, and Y. Smaragdakis. cJ: Enhancing Java with Safe Type Conditions. In *Proc. Int'l Conf. Aspect-Oriented Software Development (AOSD)*, pages 185–198. 2007.

[17] International Organization for Standardization. *ISO/IEC 9899-1999: Programming Languages—C*, 1999.

[18] H. P. Jepsen and D. Beuche. Running a Software Product Line

– Standing Still is Going Backwards. In *Proc. Int'l Software Product Line Conference (SPLC)*, pages 101–110. 2009.

[19] K. Kang, S. G. Cohen, J. A. Hess, W. E. Novak, and A. S. Peterson. Feature-Oriented Domain Analysis (FODA) Feasibility Study. Technical Report CMU/SEI-90-TR-21, SEI, 1990.

[20] C. Kästner. *Virtual Separation of Concerns*. PhD thesis, University of Magdeburg, 2010.

[21] C. Kästner and S. Apel. Type-checking Software Product Lines – A Formal Approach. In *Proc. Int'l Conf. Automated Software Engineering (ASE)*, pages 258–267. 2008.

[22] C. Kästner, S. Apel, S. Trujillo, M. Kuhlemann, and D. Batory. Guaranteeing Syntactic Correctness for all Product Line Variants: A Language-Independent Approach. In *Proc. Int'l Conf. Objects, Models, Components, Patterns (TOOLS EUROPE)*, pages 175–194. 2009.

[23] M. Latendresse. Rewrite Systems for Symbolic Evaluation of C-like Preprocessing. In *Proc. European Conf. on Software Maintenance and Reengineering (CSMR)*, pages 165–173. 2004.

[24] K. Lauenroth, K. Pohl, and S. Toehning. Model Checking of Domain Artifacts in Product Line Engineering. In *Proc. Int'l Conf. Automated Software Engineering (ASE)*, pages 269–280. 2009.

[25] J. Liebig, S. Apel, C. Lengauer, C. Kästner, and M. Schulze. An Analysis of the Variability in Forty Preprocessor-Based Software Product Lines. In *Proc. Int'l Conf. Software Engineering (ICSE)*, pages 105–114. 2010.

[26] B. McCloskey and E. Brewer. ASTEC: A New Approach to Refactoring C. In *Proc. Europ. Software Engineering Conf./Foundations of Software Engineering (ESEC/FSE)*, pages 21–30. 2005.

[27] M. Mendonça, A. Wąsowski, and K. Czarnecki. SAT-based Analysis of Feature Models is Easy. In *Proc. Int'l Software Product Line Conference (SPLC)*, pages 231–240. 2009.

[28] Y. Padioleau. Parsing C/C++ Code without Pre-Processing. In *Proc. Int'l Conf. Compiler Construction (CC)*, pages 109–125. 2009.

[29] T. T. Pearse and P. W. Oman. Experiences Developing and Maintaining Software in a Multi-Platform Environment. In *Proc. Int'l Conf. Software Maintenance (ICSM)*, pages 270–277. 1997.

[30] D. Pech, J. Knodel, R. Carbon, C. Schitter, and D. Hein. Variability Management in Small Development Organizations – Experiences and Lessons Learned from a Case Study. In *Proc. Int'l Software Product Line Conference (SPLC)*, pages 285–294. 2009.

[31] H. Post and C. Sinz. Configuration Lifting: Verification meets Software Configuration. In *Proc. Int'l Conf. Automated Software Engineering (ASE)*, pages 347–350. 2008.

[32] J. G. Refstrup. Adapting to Change: Architecture, Processes and Tools: A Closer Look at HP's Experience in Evolving the Owen Software Product Line. In *Proc. Int'l Software Product Line Conference (SPLC)*, 2009. Keynote presentation.

[33] S. She, R. Lotufo, T. Berger, A. Wąsowski, and K. Czarnecki. The Variability Model of The Linux Kernel. In *Proc. Int'l Workshop on Variability Modelling of Software-intensive Systems (VaMoS)*, pages 45–51. 2010.

[34] R. Tartler, J. Sincero, D. Lohmann, and W. Schröder-Preikschat. Efficient Extraction and Analysis of Preprocessor-Based Variability. In *Proc. Int'l Conf. Generative Programming and Component Engineering (GPCE)*, 2010.

[35] R. Tartler, J. Sincero, W. Schröder-Preikschat, and D. Lohmann. Dead or Alive: Finding Zombie Features in the Linux Kernel. In *Proc. GPCE Workshop on Feature-Oriented Software Development (FOSD)*, pages 81–86. 2009.

[36] S. Thaker, D. Batory, D. Kitchin, and W. Cook. Safe Composition of Product Lines. In *Proc. Int'l Conf. Generative Programming and Component Engineering (GPCE)*, pages 95–104. 2007.

[37] M. Vittek. Refactoring Browser with Preprocessor. In *Proc. European Conf. on Software Maintenance and Reengineering (CSMR)*, pages 101–110. 2003.

[38] D. Weise and R. Crew. Programmable Syntax Macros. In *Proc. Conf. Programming Language Design and Implementation (PLDI)*, pages 156–165. 1993.

Patching Product Line Programs[*]

Martin Kuhlemann
Faculty of Computer Science
University of Magdeburg, Germany
martin.kuhlemann@ovgu.de

Martin Sturm
Faculty of Computer Science
University of Magdeburg, Germany
MartinSturm@gmx.net

ABSTRACT

Software product line engineering is one approach to implement sets of related programs efficiently. Software product lines (SPLs) can be implemented using code transformations which are combined in order to generate a program. A code transformation may add functionality to a program or may alter its structure. Though implemented with less effort, a single malfunctioning SPL program is harder to patch because patches must effect the SPL transformations which the program was generated from. In this paper, we present a new approach to patch programs of a transformation-based SPL. We demonstrate the feasibility of this approach using a prototype.

Categories and Subject Descriptors

D.2.5 [**Software Engineering**]: Testing and Debugging—*Debugging aids*

General Terms

Algorithms, Design

1. INTRODUCTION

A *software product line (SPL)* is a set of related programs which are generated from a shared code base [6]. SPL programs are defined using *features* (user-visible program characteristics [17]) and programs of an SPL differ in features. Features can be implemented by program transformations, which add functionality to a program or alter the structure of a program. An SPL program then is generated by selecting features and executing code transformations which implement those features. Reusing transformations across SPL programs reduces the overall effort to implement the SPL programs. In prior studies on transformation-based SPLs, however, we and others observed that errors were hard to track and remove [2, 20, 33]. The solutions presented accordingly concentrate on syntactic correctness but semantic correctness is not discussed, i.e., patching SPL programs is not discussed.

As a first contribution of this paper, we provide an analysis on how SPL programs can be patched (manually edited). In particular, we compare the *patching of SPL transformations* with the *patching of generated programs*. As sample transformations used in SPLs, we concentrate on superimpositions and refactorings. From our analysis we argue that patching the *transformations* of an SPL can be inappropriate – we need techniques to also patch the *generated* code.

As a second contribution, we present and demonstrate a new approach for propagating patches from generated SPL programs to SPL transformations. That is, after the developer stepped through and patched the generated program, a propagation tool identifies and propagates the patches to the SPL transformations. In this approach, we combine an index data structure with a transformation history data structure to optimize patch propagation. We demonstrate the feasibility of this approach with our prototype.

[*]This paper summarizes and extends the Master's Thesis of Martin Sturm [31]. An extended version of this paper with more technical details has been published before as a technical report [21]. The authors thank Christian Kästner, Don Batory, and Marko Rosenmüller for helpful comments.

2. TRANSFORMATIONS IN SPLS

Superimposition. A number of approaches use superimposition transformations to implement SPLs, e.g., [2, 4, 25]. A superimposition creates classes in its input program and executes class refinements on this input program. A *class refinement* creates members in input-program classes and executes method refinements. *Method refinements* create statements in input-program methods. We use feature-oriented programming to represent SPL techniques that use superimposition transformations; further, we use Jak as a sample feature-oriented language [4]. Jak extends Java by superimposition mechanisms.

In Figure 1a, we sample code which is similar to code of an SPL from prior work where we integrated programs of a compression-library SPL with environments, i.e., we bridged incompatible structures between them [19]. A superimposition transformation *Base* is defined to transform a possibly empty input program by creating a class ZipArchive, i.e., *Base* encapsulates ZipArchive. Superimposition *Stats* encapsulates a class refinement of ZipArchive which extends ZipArchive in the input program of *Stats*. This class refinement encapsulates members (Lines 4-7) and a method refinement (Lines 8-10). The method refinement ZipArchive.getID

Transformation *Base*

```
 1 class ZipArchive{
 2   int getID(){return 1;} }
```

Transformation *Stats*

```
 3 refines class ZipArchive{
 4   int counter=0;
 5   void count(){
 6     counter = counter/1;
 7     System.out.print(counter); }
 8   int getID(){
 9     count();
10     return Super.getID(); } }
```

Transformation *MakeCompatible*

```
11 Rename method: ZipArchive.getID() ↦id
```

(a) SPL transformations.

```
1 class ZipArchive{
2   int counter=0;
3   void count(){
4     counter = counter/1;
5     System.out.print(counter);}
6   int getID(){
7     count();
8     return 1;} }
```

(b) SPL product generated from *Stats*, *Base*.

```
1 class ZipArchive{
2   int counter=0;
3   void count(){
4     counter = counter/1;
5     System.out.print(counter);}
6   int id(){
7     count();
8     return 1;} }
```

(c) SPL product generated from *MakeCompatible*, *Stats*, *Base*.

Figure 1: Superimpositions and their composition result.

of *Stats* extends method ZipArchive.getID in the input program of *Stats* (method created by *Base*) and creates statements in this method. The extended method is called with Super (Line 10). The result of executing *Base* and *Stats* from Figure 1a is shown in Figure 1b. A class ZipArchive there encapsulates the members of both ZipArchive class fragments it was generated from (members getID, counter, count). Method getID encapsulates the code generated from getID of *Base* and of *Stats*.

Refactoring. Program generation may involve configuring the program's structure using selectable refactorings. *Refactorings* are transformations which alter the structure of a program but do not alter its functionality [13]. For example, renaming a method of a program and updating all method calls is a Rename-Method refactoring [13]. We use *refactoring feature modules (RFMs)* [19] to represent SPL techniques which use refactoring transformations.

We introduced RFMs to allow configuring the structure of SPL programs [19]. RFMs allowed us to reuse SPL programs in different incompatible environments in which they could not have been reused as is before. One selection of refactorings then allowed us to reuse SPL programs in one environment and another selection allowed us to reuse *the same programs* in another environment.

In Figure 1a, we defined an RFM *MakeCompatible* in the SPL code base. Executing *Base*, *Stats* and *MakeCompatible* generates a class ZipArchive with members id, count, and counter but no member getID, see Figure 1c.

Refactoring transformations have preconditions that specify which properties a piece of code must fulfill such that the executed transformation does not alter the piece's functionality [28]. The above refactoring of *MakeCompatible* (Rename Method: ZipArchive.getID ↦ id) requires that method ZipArchive.getID exists and that no method ZipArchive.id exists in the code to refactor. (Further preconditions exist, but are not important for now.)

Refactorings can replace pieces of code one-by-one, can merge code, can multiplex code, and can create code in their respective input programs. Since transformations with these abilities are considered non-trivial [1, 8, 12, 38], refactorings are non-trivial transformations in SPL technology. For example, a Rename-Class refactoring inside an RFM replaces (renames) a class of an input program one-by-one by a class in the according output program [13]. A Pull-Up-Method refactoring inside an RFM merges multiple methods of an input program into one method in the according output program.[1] A Push-Down-Field refactoring inside an RFM multiplexes a single field of an input program in the according output program.[2] An Encapsulate-Field refactoring inside an RFM creates a get and a set method for a field and no code is removed for that [13].

3. PROBLEM STATEMENT

A generated SPL program may work incorrectly due to bugs. As an example, we placed a bug in the code of Figure 1. The program in Figure 1c should print the operation count (consecutive numbers) but it prints zeros instead. The reason is that the code in Line 4 is incorrect and should be patched to "counter=counter+1;". Finally, however, this patch must affect Line 6 of the SPL transformations (Fig. 1a). To patch SPL programs, we can (a) patch the *transformations* (Fig. 1a) and then regenerate the program, or (b) patch the *generated* program (Fig. 1c) and later possibly propagate the patches to the transformations. We now discuss strengths and weaknesses of both approaches.

3.1 Mapping Problem

SPL transformations like superimpositions and refactorings are executed one after the other such that the overall mapping of code in the generated program to code in transformations is complex [12, 36]. A piece of code in the generated program might be the result of merging, multiplexing, *and* replacing code during program generation, so we should hide this complex mapping from the developer. Additionally, the code shown to the developer should be in a language which she is familiar with.

When patching generated programs, a patch must be propagated to an SPL transformation T (we call this transformation *target*). For that, all SPL transformations $\mathbb{T}$ that follow T must be inverted (undone) in the generated program. To invert a refinement we can instantiate a remove operation; for a refactoring A, we can instantiate a refactoring of which the type is hard-coded to invert A. Patches may prevent a transformation of $\mathbb{T}$ being inverted. For example, adding a method ZipArchive.getID to the code of Figure 1c prevents inverting the transformation *MakeCompatible* (Re-

[1]Pull-Up Method moves a subclass method to a superclass and removes equivalent methods of other subclasses [13].
[2]Push-Down Field generates a copy of the pushed field in multiple subclasses and removes the superclass field [13].

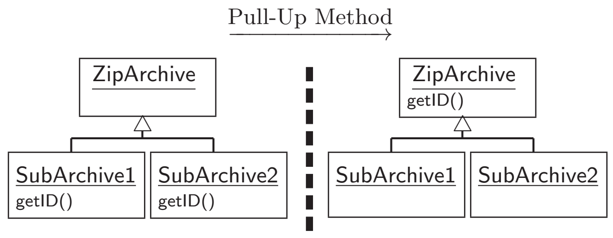


Figure 2: Pull-Up Method refactoring merges code.

name Method: ZipArchive.getID $\mapsto$ id) because otherwise ZipArchive.getID would be generated twice in SPL programs (which is an error in most languages). Approaches to patch generated programs, thus, need a *fallback strategy*. The code shown to the developer is in a language (Java) which is very similar to the language the developer used to implement the SPL (Jak+RFM), i.e., she should be familiar with it.

When patching SPL transformations and the code executed in the generated program was created by a refactoring (code did not exist before), there is no code on the level of the transformations to show to a developer accordingly [38]. Showing transformation definitions instead is not an option as they might not show the code they generate, either. For example, RFMs define properties of code to generate but RFMs do not include this code [19], e.g., they define which field to encapsulate but do not include get and set methods. However, skipping this code is error prone [12].

If an executed method was merged by a refactoring from multiple methods, each of these methods in the transformations is a valid (with respect to the generated program) mapping value from the executed, generated method [38]. Showing the wrong method with respect to the transformations, however, causes confusion. In the example of Figure 2, a Pull-Up-Method refactoring merges SubArchive1.getID and SubArchive2.getID to ZipArchive.getID. Assume, the patch tool defines SubArchive1.getID to represent the executed, merged method ZipArchive.getID on the level of the transformations. But if SubArchive2.getID was called on the level of transformations the developer is confused seeing SubArchive1.getID.

If executed code was merged by a refactoring from multiple methods, then breakpoints set in the transformations to one of the original methods can match too often or too rarely [1]. For example, methods SubArchive1.getID and SubArchive2.getID in Figure 2 are merged by a Pull-Up-Method refactoring. Assume, the patch tool maps the executed, merged method ZipArchive.getID to SubArchive1.getID which hosts a breakpoint in the transformations, then this breakpoint will match for SubArchive2.getID, too incorrectly. A breakpoint set to SubArchive2.getID (not referenced from the generated program) will never match. This nondeterminism hampers stepping through the program.

If executed code was generated by a refactoring which multiplexes code, multiple values of variables from the generated program might need to be merged on the level of transformations – this, however, might be impossible. As an example, consider a Push-Down-Field refactoring on a static field which generates multiple static fields in the generated program; these generated fields can expose different values in the executed program but can only expose one on the level of transformations. According values must be merged to be presented as one value for the single (pushed) field in the transformations. However, this is not possible generally. Variable values then cannot be analyzed.

The code shown to the developer is in the language the developer used to write the SPL, i.e., she should be familiar with it. (Solutions exist for problems similar to the problems above [1, 14, 38]. However, approaches are limited to languages for which conditional breakpoints and path analyses can be compiled into binaries.)

3.2 Scattering Program Code

During patching an SPL program a developer should concentrate on this program's bug in the first place. Just in the second place other SPL programs should be considered.

When patching generated programs, code of other SPL programs, i.e., code that does not contribute to the patched program, is hidden and errors in these other programs are postponed until finishing program patching. Nevertheless, the patches can be checked automatically against all SPL programs, e.g., during propagation, and an error can be reported when a patch introduces a program in error [33].

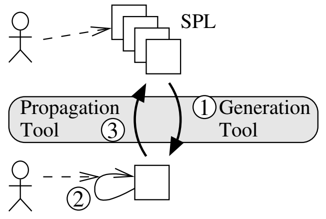
When patching transformations, code that contributes to one SPL program is scattered across transformations. For that, the developer must execute the transformations *in-mind* in order to foresee the code actually executed when stepping over it, e.g., when stepping over a method call. The developer needs similar knowledge to decide which methods she can call in a patch of a transformation ("How do input programs of a transformation-to-patch look like?"). Note, if tools would visualize generated code they switch to the patch-generated-programs approach.

As an example of in-mind transformation execution, reconsider Figure 1a. In these SPL transformations, a developer may want to add to superimposition *Base* a method that calls Element.id (not depicted). As Element.id is undefined in *Base* the developer must generate the patched program in-mind to verify that she is able to call this method in *Base*. Transformations in the SPL code base which do not contribute to the patched program distract the developer from the bug to repair.

Code in unpatched superimpositions might be replaced or overridden by a patch, accidentally, in the currently patched SPL program or others. For example, when a developer applies a patch (not depicted) in Figure 1a such that she adds a new superimposition *AfterStats* which follows *Stats* and such that she adds a method ZipArchive.count to *AfterStats*, then this *AfterStats* method replaces ZipArchive.count of *Stats*. It can get worse. If *Stats* would not contribute to the patched program but to a different program, the patch could replace count in this other program, unnoticed. Current approaches which validate the whole SPL [33] may help but executing them during patching hampers implementing the patch. When a method created in a patch *overrides* a method in a different SPL program accidentally current mechanisms do not help; especially, when the developer intended to override some different method.

3.3 Bounded Quantification Problem

Bounded quantification is a guideline to reduce complexity in transformation systems [23, 26]. Bounded quantification restricts code generated by a transformation to only access code which exists in the transformation's input program, i.e., which has been generated by preceding transformations.



①=select SPL program
②=patch SPL program
③=propagate patches

Figure 3: Use case for patching SPL programs.

Bounded quantification should hold before patching and after, e.g., no developer/tool should patch a class in a transformation A to subtype a class which is generated by a follower transformation of A.[3] Developers should be advised in which transformation and how to implement a patch.

When patching generated programs, a patch is detected and propagated automatically. Thereby, the tool can advise in choosing a transformation to host the patch (target). The tool can further check that bounded quantification is not broken with this advice, even in other SPL programs.

When patching transformations, the developer may realize during patching that with the patch just implemented she breaks bounded quantification. Then she might have to move the patch (manually) into another transformation. Additional complexity is put upon the developer when patches break *subsequent* transformations (maybe in other SPL programs). In the example of Figure 1a, a developer, who aims to patch superimposition *Base* and, for that, adds a method ZipArchive.id, breaks the follower RFM *MakeCompatible* because this RFM requires ZipArchive.id not to exist in its input program. With a growing number of superimposition and refactoring transformations, the restrictions imposed by transformations, which follow a transformation to patch, become opaque and unmanageable [12].

3.4 Summary

Patching SPL transformations may present incorrect or no code to the developer (cf. Sec. 3.1), may show scattered code (cf. Sec 3.2), and may require to re-implement patches (cf. Sec. 3.3). Patching generated programs may "just" require a fallback strategy when transformations cannot be inverted.

4. PATCHING THE TRANSFORMATIONS

Some researchers argue to step through code and to patch it on the level of transformations (there: high-level code), e.g., [7, 15, 37]. From our analysis and in line with others [1, 8, 12, 36] we argue that stepping through code can be meaningful at every level of abstraction. We show now how code generated from transformations can be patched, too.

4.1 Conceptual Process

We propose to let the developer step through and patch the generated program, and to propagate *automatically* each patch to the best SPL transformation; this use case is depicted abstractly in Figure 3. The propagation tool is proposed to *find and link patches*, *prepare the propagation*, *perform the propagation*, and to *save the propagation*.

[3]Some languages used for SPLs, e.g., Jak or AspectJ, do not enforce bounded quantification. Patches to SPLs written in these languages can break bounded quantification.

Index Key	Index Value
ZipArchive.id	[*Base::*ZipArchive.getID, *Stats::*ZipArchive.getID]
ZipArchive	[*Base::*ZipArchive, *Stats::*ZipArchive]
ZipArchive.counter	[*Base::*ZipArchive.counter]
ZipArchive.count	[*Stats::*ZipArchive.count]

Table 1: Sample index for code of Figure 1c.

Find and link patches. At first, the propagation tool should compare the patched program with an unpatched version of this program to find patches. To ease propagation later, the tool should link each patch to a qname (abbreviation for fully-qualified name) which encapsulates the patch. In our example of Figure 1, the propagation tool should find the patch in Line 4 of Figure 1c and should link it to qname ZipArchive.count.

Prepare propagation. To advise *where*, i.e., to which SPL transformation, to propagate a patch best later (we call such transformation *target*), the propagation tool should calculate the origins of generated members and classes. For that, it should analyze the executed superimposition transformations and refactoring transformations in execution order. The propagation tool should record in an *index structure* for every generated qname a (list of) qname from the transformations of which the code includes the generated code. If there is no such qname in the transformations, i.e., when an RFM creates code, then an empty list is recorded as index value for the qname of this code.

An index for the SPL program of Figure 1c is given in Table 1. The index key is a qname from the generated program and the index value is a list of qnames from code of the transformations, prefixed with these transformations. For example, ZipArchive.id is indexed to be generated from ZipArchive.getID in transformations *Base* and *Stats*.

To advise *how* a patch should be implemented in the target later, the propagation tool should record all code transformations which executed in a *transformation history*. In the transformation history of Figure 1c, *MakeCompatible* is recorded to affect *Base* and *Stats*.

Perform propagation. To perform patch propagation, the propagation tool first should calculate a good target and then invert all transformations on the patch which executed after the target during program generation. The tool should calculate a good target in three steps: First, the tool analyzes the qnames which the patch relates/references to, e.g., qnames of called method and hosting classes – the origin of these qnames is the best target for hosting the patch. Second, to avoid that an inverted transformation cannot be reperformed after propagation, the propagation tool should, for every transformation to invert, analyze the qnames and their relations in the respective transformation's input program. That is, the described index should also keep qnames deleted during program generation and tag them. If qname relations required by a transformation conflict a patch (disallow to invert a transformation), then we apply a *fallback strategy*. Third, the propagation tool should check whether bounded quantification is broken when the patch is propagated to its target – if so, the propagation tool should adapt the target to be the *follower* of all transformations which introduce code the patch references (required condition for bounded quantification).

The fallback strategy we propose (there might be more): When inverting a transformation fails, the propagation tool creates a transformation which follows the non-invertible transformation; a transformation which will replace the erroneous code in future SPL programs.

We want to exemplify patch propagation with the priorly discussed patch for Figure 1c ("counter=counter/1;" $\mapsto$ "counter=counter+1;"). The patch is found and linked to ZipArchive.count. In the index shown in Table 1, ZipArchive.count is recorded to be generated in *Stats*, so *Stats* is calculated as target for the patch. The patch does not reference other qnames (bounded quantification cannot break) and so the target remains *Stats*. The propagation tool finally inverts *MakeCompatible* on the patch (nothing changes) and provides this propagated patch as advice to the developer. If *MakeCompatible* could not be inverted in the presence of patched ZipArchive.count, the tool should advise our fallback strategy, i.e., it should insert a new transformation as a follower of *MakeCompatible* with patched ZipArchive.count.

When a qname exists in the index but maps to an empty list of value qnames (the piece of code got created by a refactoring), then the tool should propagate the patch along the reverse global sequence of program transformations. Thereby, it should invert SPL transformations until one transformation identifies the patched piece of code as "self-generated" and provides a target. As an example, assume the patched ZipArchive.count would have been created by *MakeCompatible* (e.g., when *MakeCompatible* is an Encapsulate Field refactoring) – then ZipArchive.count would occur in the index but would map to an empty list. In this case, the transformations from *MakeCompatible* to *Base* would be inverted. The inversion process of *MakeCompatible* would stop this target-less propagation because at first it defines a new target for the patch. In this special case of *MakeCompatible* creating count, the target would be a new follower of *MakeCompatible* because RFM *MakeCompatible* does not encapsulate code to patch. If *MakeCompatible* would have merged ZipArchive.count from multiple pieces of code, then the inversion algorithm for *MakeCompatible* would generate multiple targets and propagation continues.

Save propagation. We use metrics (origins of referenced code) to identify the best target for a patch and so this target might be suboptimal semantically though meaningful and correct syntactically. A propagation tool, thus, should advise a mapping to the developer but should ask the developer to confirm. For instance, the tool could save the patched SPL transformations separately such that the developer can decide whether to accept the propagation. If she accepts, the original SPL transformations are replaced.

4.2 Unsupported Patches & Transformations

The transformations available for program generation limit the patches which can be propagated from the generated program to the transformations. Depending on the transformation language, no fallback strategy seems available.

As a first example, superimpositions in Jak cannot replace constructors generated by preceding transformations [29]. Patches in constructor bodies *must* be propagated to the transformation which created the constructor. If they cannot be propagated, we emphasize the advice from the Jak documentation to *extract constructors into initialization methods* [29]; methods which then can be replaced with our fallback strategy.[4]

As a second example, code removed from the generated program might not be allowed to be removed in the transformations, accordingly. If the deletion in the generated program prevents to invert a transformation, a propagation tool cannot propagate the patch but also cannot delete the qname by adding a superimposition (because superimpositions can only generate code) nor by adding a refactoring (because refactorings can only change code structure).

Refactorings (e.g., of RFMs), refinements (e.g., of Jak) as well as aspects [18] and rewrites [35] can be inverted such that patch propagation is possible, but this is not the general case. Higher-order rewrites (where pieces of code are matched by incomplete patterns) [34] can only be inverted when code matched with wildcards can be reconstructed. If a transformation cannot be inverted, we propose to apply our fallback strategy.

The mapping from the generated program towards the transformations is specific to tools. If different tools translate the same set of transformations differently, the index creation tool must be parameterized with the transformation tool used. For example, for Jak at least 2 different tools with 2 different translations into Java exist [3]. (Note, this situation adds effort in every solution approach.)

5. PROTOTYPE & DEMONSTRATION

We demonstrate that stepping through and patching the generated program and that propagating patches afterwards *is feasible* for superimposition- and refactoring-based SPLs. That is, we implemented the above concepts prototypically and used the tool in a demonstrating example.

Our prototype finds patches and links them to qnames (Phase 2 in Sec. 4.1), calculates the best target for every patch (Phase 3), and stores the propagation advice separately (Phase 4). To do this, for every superimposition (analyzed in composition order), the prototype collects the qnames of code which is generated in these superimpositions as index keys and as index values. Recorded transformations are executed on index keys but not on index values. Thus, the index in the end maps qnames of the generated program to qnames of the transformations. Our prototype currently supports one transformation tool for Jak and one for refactoring feature modules (RFMs) – error detection based on qname *relations* during program generation is not yet implemented (keeping and tagging deleted qnames). For more implementation details, simplifications, and for more demonstrating examples please consult [21, 31] – they are omitted for readability.

Demonstrating example. We demonstrate the propagation approach using the *Graph Product Line (GPL)* which has been proposed to be a standard benchmark for SPL technology [22]. Specifically, we use a version from prior work in which we added RFMs to GPL in order to integrate GPL

[4]For patched field initializations (fields also cannot be replaced in Jak superimpositions) we envision to encapsulate their initialization in methods, too, which can be replaced.

```
                                   Transformation (Superimp.) Directed
 1 class Graph{ ...
 2   public void addEdge(Edge the_edge){
 3     Vertex start = the_edge.start;
 4     Vertex end = the_edge.end;
 5     edges.add( the_edge );
 6     start.addNeighbor(new Neighbor(end,the_edge)); }}

                                   Transformation (Superimp.) Weighted
                                   Transformation (Superimp.) Shortest
                                   Transformation (Superimp.) Benchmark
                                   Transformation (Superimp.) adaptToClient
                                   Transformation (RFM) VertexVerteximpl
 7 Rename class: Vertex ↦VertexImpl

                                   Transformation (RFM) GraphWgraph
 8 Rename class: Graph ↦WeightedGraphImpl

                                   Transformation (RFM) AddvertexAdd
 9 Rename method:
     WeightedGraphImpl.addVertex(VertexImpl) ↦add

                                   Transformation (RFM) ShortestSmall
10 Rename method:
     WeightedGraphImpl.ShortestPath(VertexImpl) ↦shortestPath
```

(a) Original SPL transformations

$\Downarrow$

```
 1 class WeightedGraphImpl{ ...
 2   public void addEdge(Edge the_edge) {
 3     VertexImpl start = the_edge.start;
 4     VertexImpl end = the_edge.end;
 5     edges.add(the_edge);
 6     start.addNeighbor(new Neighbor(end, the_edge)); }}
```

(b) Generated SPL program

$\Rightarrow$

```
 1 class WeightedGraphImpl{ ...
 2   public void addEdge(Edge the_edge) {
 3     if (the_edge != null) {
 4       VertexImpl start = the_edge.start;
 5       VertexImpl end = the_edge.end;
 6       edges.add(the_edge);
 7       start.addNeighbor(new Neighbor(end, the_edge));
 8     }else{
 9       System.out.println("Param the_edge was null!"); } }
10   public WeightedGraphImpl ShortestPath(VertexImpl s) {
11     return shortestPath(s); }}
```

(c) Patched, generated SPL program

$\Uparrow$

```
                                   Transformation (Superimp.) Directed
 1 class Graph{ ...
 2   public void addEdge(Edge the_edge) {
 3     if(the_edge != null){
 4       Vertex start = the_edge.start;
 5       Vertex end = the_edge.end;
 6       edges.add(the_edge);
 7       start.addNeighbor(new Neighbor(end, the_edge));
 8     }else{
 9       System.out.println("Param the_edge was null!"); } }}

            Transformation (Superimp.) notInvertibleFor_ShortestSmall
10 refines class WeightedGraphImpl{
11   public WeightedGraphImpl ShortestPath(VertexImpl s){
12     return shortestPath(s); }}
```

(d) Propagation advice (patched SPL transformations)

Figure 4: Stepping through and patching the code of the GPL class Graph/WeightedGraphImpl.

programs with incompatible environments [19].[5] The refactorings we apply, Rename-Class and Rename-Method, pose an important fraction when integrating programs [19].

We selected 9 features from GPL which correspond to program transformations which in turn generate a compilable program; 5 superimpositions and 4 RFMs. In Figure 4a, we list the transformation names in execution order (top-down) and show relevant code snippets from these transformations. The superimpositions generate and refine the classes Edge, Graph, Neighbor, and Vertex. The RFMs rename class Vertex into VertexImpl, class Graph into WeightedGraphImpl, method WeightedGraphImpl.addVertex(VertexImpl) into add, and method WeightedGraphImpl.ShortestPath(VertexImpl) into shortestPath. With these RFMs, the GPL program of the 4 superimpositions can be configured to be reusable as a library in a program with which this GPL program was incompatible before [19]. As we did not find bugs in the GPL program, we patched this generated program at will to cover interesting cases.

Using Figure 4, we want to demonstrate the proposed process in a practical scenario for patching class Graph. At first the developer executes the SPL transformations (Fig. 4a) to generate the SPL program (Fig. 4b). Later, this program is observed to not work properly, is stepped through, and is patched (Fig. 4c). By comparing the generated program (Fig. 4b) and the generated, patched code (Fig. 4c), the propagation tool finds three patches applied to class WeightedGraphImpl – we underline them in Figure 4c. The tool links these patches to qnames, e.g., it links the patch in Line 3 to WeightedGraphImpl.addEdge(Edge). The method addEdge(Edge) is found to be patched (not created) because an index key exists. As a result, the tool computes from its index that addEdge was generated lastly in the superimposition *Directed* which becomes target (cf. Fig. 4a). The propagation tool detects that code in the patch does not reference qnames others than the hosting method addEdge did

[5]We pruned the GPL version according to the current limitations of our prototype.

```
 1 class VertexImpl{ ...
 2  private boolean displayed = false;
 3  public void display() {
 4    System.out.print("Pred " + predecessor + " DWeight " +
        dweight + " ");
 5    display$$eval$outWeighted$GG();
 6    this.displayed = true;}
 7  public boolean wasDisplayed(){
 8    return displayed; }
 9  public VertexImpl assignName(String name) {
10    this.name = name;
11    if(this.wasDisplayed()){
12      System.out.println("was already displayed!"); }
13    return (VertexImpl)this; }}
```

Figure 5: Patches to Vertex/VertexImpl.

before, so *Directed* remains target. The propagation tool inverts the 4 RFMs, which executed after target *Directed*, on the patch and advises the developer to replace method addEdge in transformation *Directed* (Fig. 4d, Lines 2-9). The second patch (Fig. 4c, Lines 8-9) is linked to the same qname WeightedGraphImpl.addEdge and is propagated together with the first one.

The third patch (Fig. 4c, Lines 10-11) requires the fallback strategy. The patch concerns a method ShortestPath, which was created during patching (no index key exists). The tool analyzes that ShortestPath solely references shortestPath (generated in *Shortest*) and thus the tool uses *Shortest* as target for ShortestPath. Next, the target *Shortest* is validated whether all RFMs that executed after *Shortest* can be inverted with ShortestPath.[6] The propagation tool cannot invert RFM *ShortestSmall* (Rename Method: WeightedGraphImpl.ShortestPath(VertexImpl) $\mapsto$ shortestPath) because this would make SPL transformations, which execute before *ShortestSmall*, generate two methods WeightedGraphImpl.ShortestPath(VertexImpl) in products – this is an error in most product languages. Following our fallback strategy, the propagation tool adds a superimposition *notInvertibleFor_ShortestSmall* as a follower of *ShortestSmall* (cf. Fig. 4d), a superimposition which then adds the patch ShortestPath to future programs. Note in Figure 4d, as *ShortestSmall* is not inverted, the tool advises to refine class WeightedGraphImpl (exists after *ShortestSmall*) instead of Graph.

In a final analyzed case, we added references towards qnames in a patch. In Figure 5, we underline the patches applied to class VertexImpl.[7] The field displayed and the method wasDisplayed got added, and methods display and assignName got patched to access the added field and method. Our propagation tool detects those accesses and for that advises to use superimposition *Shortest* (refines display lastly) as target for displayed. As wasDisplayed got added and solely accesses displayed, the propagation tool advises to propagate wasDisplayed to *Shortest* (target of displayed), too. If we would patch display and assignName in *Directed* (creates display and assignName), the reference to a *Shortest* method or field would break bounded quantification. For that, the patches to methods assignName and display are advised to be propagated to *Shortest*, too. In *Shortest* they replace the methods created in *Directed*. Summarizing, our tool advised well *where and how* to propagate patches which we made to an SPL program.

6. RELATED WORK

There is much work on how to relate generated and transformation code, e.g., for stepping through code [12, 27, 30, 32, 34, 35, 37]. In addition to this work, we *propagate* patches from generated SPL programs to the code base of an SPL. There is further work on how to propagate patches from generated programs to the code of a superimposition-based SPL [3]. In addition to this work, we support SPLs which are implemented by superimpositions *and refactorings*.

Compilers execute refactoring-like transformations (optimizations) on code which keep functionality. Patching these (one-of-a-kind) programs poses similar problems as we faced for SPLs [1, 7, 14, 15, 36, 38]. In contrast to according work, the transformations we considered (superimpositions and refactorings) cannot only change code structure but also add functionality. As a result, executed code exists in the generated program but has no origin in the (possibly empty) initial input program.

MolhadoRef inverts refactorings to reduce human interaction when integrating a patched program with a former revision of this program [10, 11]. Lynagh provides ideas similar to MolhadoRef for edits [24]. While both approaches propagate patches of a program toward a single program (an earlier revision), we propagate patches of a program toward the code base of a transformation-based SPL.

Bidirectional transformations (a.k.a. lenses) synchronize multiple related representations of elements where patches can be propagated in any representation [9, 16]. In the patch propagation problem we focused on, edits to the generated program may prevent the execution of inverse RFMs (we, thus, discussed a fallback strategy) – such situation may not occur for bidirectional transformations.

Design maintenance systems execute transformations in order to generate a program [5]. In design maintenance systems, maintenance deltas are transformations which are added to a transformation history during maintenance of the generated program [5]. Baxter indicates patch propagation toward an old abstraction (specification) [5], but this old abstraction is no code base of an SPL (instrumenting superimpositions and refactorings). Baxter did not explore backward integration mechanisms though.

7. CONCLUSIONS

In this paper we discussed a number of problems which occur when stepping through and patching a program generated from a transformation-based software product line (SPL). Specifically, we discussed problems of complex mappings of code between the transformations and the generated program, problems of scattered SPL program code, and problems of patches that increase complexity. We found that for SPLs implemented with transformations of superimpositions and refactorings, the *generated* code is a beneficial option for stepping and patching. We automated the propagation of patches from the generated program to the SPL transformations and demonstrated its feasibility.

[6]Superimpositions are invertible in Jak naturally due to the Jak composer implementation. In other languages local variables might need to be transformed into fields.

[7]Line 5 of Fig. 5 shows a possible translation for Super.display(); inlining this call would remove the statement.

For our approach, we combined index techniques and transformation histories (both known from other contexts) to aid patching in transformation-based SPLs. Our propagation tool links detected patches to fully-qualified names of the generated program. It calculates the best SPL transformation to host a patch. The tool finally advises *how* to integrate the patch with the SPL transformations.

8. REFERENCES

[1] A.-R. Adl-Tabatabai. *Source-level debugging of globally optimized code*. PhD thesis, Carnegie Mellon University Pittsburgh, 1996.

[2] S. Apel, C. Kästner, A. Größlinger, and C. Lengauer. Type safety for feature-oriented product lines. *Automated Software Engineering – An International Journal*, 17(3):251–300, 2010.

[3] D. Batory. A tutorial on feature oriented programming and the AHEAD tool suite. In *GTTSE*, pages 3–35, 2006.

[4] D. Batory, J.N. Sarvela, and A. Rauschmayer. Scaling step-wise refinement. *TSE*, 30(6):355–371, 2004.

[5] I.D. Baxter. *Transformational maintenance by reuse of design histories*. PhD thesis, University of California at Irvine, 1990.

[6] P. Clements and L. Northrop. *Software product lines : Practices and patterns*. Addison-Wesley, 2006.

[7] D.L. Curreri, A.K. Iyengar, R.A. Biesele, and M.A. Ruscetta. Debugging optimized code using data change points, 2000. US patent #6,091,896.

[8] K. Czarnecki and U. Eisenecker. *Generative programming: Methods, tools, and applications*. Addison-Wesley, 2000.

[9] K. Czarnecki, J.N. Foster, Z. Hu, R. Lämmel, A. Schürr, and J.F. Terwilliger. Bidirectional transformations: A cross-discipline perspective. In *ICMT*, pages 260–283, 2009.

[10] D. Dig. *Automated upgrading of component-based applications*. PhD thesis, University of Illinois at Urbana-Champaign, 2007.

[11] D. Dig, K. Manzoor, R.E. Johnson, and T.N. Nguyen. Effective software merging in the presence of object-oriented refactorings. *TSE*, 34(3):321–335, 2008.

[12] R.E. Faith. *Debugging programs after structure-changing transformation*. PhD thesis, University of North Carolina at Chapel Hill, 1998.

[13] M. Fowler. *Refactoring: Improving the design of existing code*. Addison-Wesley Longman Publishing Co., Inc., 1999.

[14] J. Hennessy. Symbolic debugging of optimized code. *TOPLAS*, 4(3):323–344, 1982.

[15] U. Hölzle, C. Chambers, and D. Ungar. Debugging optimized code with dynamic deoptimization. *ACM SIGPLAN Notices*, 27(7):32–43, 1992.

[16] Z. Hu, S.-C. Mu, and M. Takeichi. A programmable editor for developing structured documents based on bidirectional transformations. In *PEPM*, pages 178–189, 2004.

[17] 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 Pittsburgh, 1990.

[18] G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, and W. G. Griswold. An overview of AspectJ. In *ECOOP*, pages 327–353, 2001.

[19] M. Kuhlemann, D. Batory, and S. Apel. Refactoring feature modules. In *ICSR*, pages 106–115, 2009.

[20] M. Kuhlemann, D. Batory, and C. Kästner. Safe composition of non-monotonic features. In *GPCE*, pages 177–186, 2009.

[21] M. Kuhlemann and M.Sturm. Debugging product line programs. Technical Report 6, Faculty of Computer Science, University of Magdeburg, 2010.

[22] R.E. Lopez-Herrejon and D. Batory. A standard problem for evaluating product-line methodologies. In *GCSE*, pages 10–24, 2001.

[23] R.E. Lopez-Herrejon and D. Batory. Improving incremental development in AspectJ by bounding quantification. In *SPLAT*, 2005.

[24] I. Lynagh. An algebra of patches, 2006. http://-urchin.earth.li/~ian/conflictors/paper-2006-10-30.pdf.

[25] M. Odersky. *The Scala language specification (version 2.7)*, 2005.

[26] D.L. Parnas. Designing software for ease of extension and contraction. In *ICSE*, pages 264–277, 1978.

[27] Z. Porkoláb, J. Mihalicza, and Á. Sipos. Debugging C++ template metaprograms. In *GPCE*, pages 255–264, 2006.

[28] D.B. Roberts. *Practical analysis for refactoring*. PhD thesis, University of Illinois at Urbana-Champaign, 1999.

[29] Software Systems Generator Research Group. *The jampack composition tool*. AHEAD tool suite v2008.07.22, manual.

[30] B. Stroustrup. *The C++ programming language*. Addison-Wesley Longman Publishing Co., Inc., 2 edition, 1991.

[31] M. Sturm. Debugging Generierter Software nach Anwendung von Refactorings. Master thesis, University of Magdeburg, Germany, 2010. http://wwwiti.cs.uni-magdeburg.de/iti_db/publikationen/ps/auto/thesisSturm.pdf.

[32] Sun Microsystems, Inc. *JSR-000045 Debugging support for other languages 1.0 FR*, 2003.

[33] S. Thaker, D. Batory, D. Kitchin, and W. Cook. Safe composition of product lines. In *GPCE*, pages 95–104, 2007.

[34] A. van Deursen and T.B. Dinesh. Origin tracking for higher-order term rewriting systems. In *HOA*, pages 76–95, 1994.

[35] A. van Deursen, P. Klint, and F. Tip. Origin tracking. *Journal of Symbolic Computation*, 15(5-6):523–545, 1993.

[36] H. Venturini, F. Riss, J.-C. Fernandez, and M. Santana. A fully-non-transparent approach to the code location problem. In *SCOPES*, pages 61–68, 2008.

[37] H. Wu, J. Gray, and M. Mernik. Grammar-driven generation of domain-specific language debuggers. *SP&E*, 38(10):1073–1103, 2008.

[38] P.T. Zellweger. An interactive high-level debugger for control-flow optimized programs (summary). In *Software Engineering Symposium on High-Level Debugging*, pages 159–171, 1983.

Implementing Feature Variability for Models and Code with Projectional Language Workbenches

Markus Voelter

Independent/itemis

Oetztaler Strasse 38,
70327 Stuttgart, Germany
voelter@acm.org

Abstract

Product line engineering deals with managing and implementing the variability among a set of related products. We distinguish between two kinds of variability: configuration and customization. Customization variability can be described using programming language code or creative construction DSLs, whereas configuration variability is described using configuration based approaches, such as feature models. Many product lines have both kinds of variability, and they need to be integrated efficiently. This paper describes an approach for product line engineering using projectional language workbenches. These represent code and models with the same fundamental technology, enabling the mixing of models and code. They make the tight integration between several domain-specific languages possible and simple. Since they can store arbitrary information in models, it is possible to overlay configuration variability over customization variability (i.e. apply feature model-based configuration to code and models). Because of the projectional approach to editing, programs can be shown with or without the dependencies on feature models, they can even be rendered (and edited) for a specific variant. This approach leads to highly integrated and productive tools for product line development. The paper explains the approach, outlines the implementation of a prototype tool based on Jetbrains MPS and illustrates the benefits using a small product line for embedded systems.

Keywords Product Line Engineering, Feature Modeling, Domain-Specific Languages, Language Composition
Classification**:** D.1.2 Automatic Programming, D.2.11 Software Architectures, D.2.3 Coding Tools and Techniques (Program editors), D.2.6 Programming Environments (Programmer workbench)

1. Introduction

The technical implementation of product line engineering focuses on two main issues: a mapping from the problem space to the solution space as well as the management and implementation of variability. In both contexts, domain specific languages (DSLs), i.e. languages that are custom-built to express specific, limited aspects of a (software) system, can help.

When configuring a product, all variation points defined in the product line have to be bound to a variant. Variation points can be bound at different times (for example, when writing the code, during system initialization, or at runtime). A variation point can also vary in the degree to which it can be configured. Two fundamental kinds of variability can be distinguished: customization and configuration.

When binding a configuration variation point, one among several alternatives is chosen. Feature models [11] are a way to describe the configuration options for a set of variation points as well as the constraints between them (such as "feature *A* cannot be selected together with feature *B"*). The number of alternative configurations may be large, but it is bounded, because only a limited number of valid feature combinations exists.

Customization variability is unbounded. A customization variation point is bound by writing a (potentially very small) program in a (perhaps very specific) language. For example, in a framework, a variation point may require the implementation of a class that implements an interface supplied by the framework, or in a data management application, a variation point may expect a regular expression that validates some data. The regular expression example suggests that it may be a good idea to define a domain-specific language (DSL) to be used to bind the variation point. DSLs usually allow the specification of an unlimited number of programs ("you can always add one more box "), but the nature of the programs is defined by the DSL.

In this paper I show how projectional language workbenches (explained in the next section) can be used for product line development, representing configuration and customization variability in the same environmnet. I argue the benefits of using DSLs to bind customization variability and will briefly show how to define DSLs and how configuration variability can be overlaid over arbitrary languages, general-purpose and domain-specific. I also show

how we can layer several languages on top of each other to enable an effective problem space to solution space mapping. An embedded systems product line will be used as the example, and the tooling will be based on JetBrains MPS [21], an Open Source projectional language workbench.

Section 2 describes the basics of projectional language workbenches in general, and MPS specifically work. Section 3 explains a feature called language annotation that is very useful for configuration variability. Section 4 shows our proof-of-concept, section 5 takes a look at future work. Section 6 puts our contribution in relationship to others', and section 7 contains a brief conclusion.

2. Projectional Language Workbenches and MPS

The term Language Workbench has been coined by Martin Fowler in 2005 [1]. In this article he characterizes it as a tool with the following properties:

- Users can freely define languages which are fully integrated with each other.
- The primary source of information is a persistent abstract representation.
- A DSL is defined in three main parts: schema, editor(s), and generator(s).
- Language users manipulate a DSL through a projectional editor.
- A language workbench can persist incomplete or contradictory information.

Projectional editing implies that all text, symbols, and graphics are projected, well-known from graphical modeling tools (UML, ER, State Charts): the model is stored independent of its concrete syntax, only the model structure is persisted, often using XML or a database. For editing purposes this abstract syntax is projected using graphical shapes. Users perform mouse and keyboard actions tailored to graphical editing to modify the abstract model structure directly. While the concrete syntax of the model does not have to be stored because it is specified as part of language definition and hence known by the projection engine, graphical modeling tools usually also store information about the visual layout.

Projectional editing can also be used for a syntax that is textual or semi-graphical (mathematical notations, for example). However, since the projection looks like text, users expect interaction patterns and gestures known from "real text" to work (such as cursor movements, inserting/deleting characters, rearranging text, selection). A projectional editor has to "simulate" these interaction patterns to be usable.

The following list shows the benefits of the approach:

- In projectional editing, no grammar or parser is used. Editing directly changes the program structure (AST). Thus, projectional editors can handle unparseable code. Language composition is easily possible, because composed languages cannot result in ambiguous grammars, a significant issue in classical parser-based systems.
- Notations are more flexible than ASCII/ANSI/Unicode. Graphical, semi-graphical and textual notations can be mixed and combined. For example, a graphical tool for editing state machines can embed a textual expression language for editing the guard conditions on transitions.
- Projectional languages by definition need an IDE for editing (it has to do the projection!), so language definition and extension always implies IDE definition and extension. The IDE will provide code completion, error checking and syntax highlighting for all languages, even when they are combined.
- Because the model is stored independent of its concrete notation, it is possible to represent the same model in different ways simply by providing several projections. Different viewpoints [23] of the overall program can be stored in one model; editing can be viewpoint or aspect specific. It is also possible to store out-of-band data, i.e. annotations on the core model/program. Examples of this include documentation, pointers to requirements (traceability) [5] or feature dependencies [6] - as we will describe below.

As a side effect, language workbenches deliver on the promise of removing the distinction between what is traditionally called programming and what is traditionally called modeling. This distinction is arbitrary: developers want to express different concerns of software systems with abstractions and notations suitable to that particular concern, formally enough for automatic processing or translation, and with good IDE support. Projectional language workbenches deliver on this goal in an integrated, consistent and productive way. They do this by applying the technology known from modeling tools (projection) to editing any notation.

The Jetbrains Meta Programming System

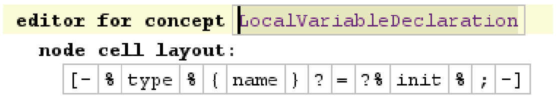
JetBrains' Meta Programming System is an open source projectional language workbench [21]. Defining a language starts by defining the abstract syntax, the editor for the language concepts is specified in a second step. Lastly the generator is defined. It outputs text (for a low-level language) or it transforms higher-level code into code expressed in lower level languages. The higher-level to lower-level generators are not text generators, they transform abstract syntax trees.

Editing the tree as opposed to "real text" needs some accustomization. Without specific adaptations, every program element has to be selected from a drop-down list and "instantiated". However, MPS provides editor customizations to enable editing that resembles modern IDEs that use automatically expanding code templates. In some cases though, the tree shines through: Consider changing a statement like *int i = j+k*; to *int i = (j+k)*2*; you cannot simply move the cursor to the left of *j* and insert a left parenthesis. Rather, you have to select the + operator (the root node of the expression on the right) and use a *Surround with Parens* refactoring. Using (as the hotkey for this refactoring creates an editing experience very similar to "real" text).

Language Definition with MPS

I have described language creation, extension and composition in MPS in a separate paper [22]. This section shows an example as a short summary. MPS, like other language workbenches, comes with a set of DSLs for language definition, a separate DSL for each language aspect. Language aspects include structure, editor, type system, generator as well as support for features such as quick fixes or refactorings.

Defining a new language starts by defining the language structure (aka meta model). This is very much like object oriented programming as language elements are represented as concepts that have properties, children and references to other concepts. The second step is the editor for the language concepts. An editor defines how the syntax for the concepts should look like - it constitutes the projection rules. Figure 1 is an example.



```
editor for concept LocalVariableDeclaration
  node cell layout:
    [- % type % { name } ? = ?% init % ; -]
```

Figure 1. Defining an editor for a local variable declaration statement (as in int i = 2*2;)

Next is the definition of the type system. For example, the type property of a *LocalVariableDeclaration* must be compatible with the type of its *init* expression.

At this point, the definition of the language and the basic editor, as well as the type system are finished. However, to use the new *LocalVariableDeclaration* statement, the user has to bring up the code completion menu in the editor, select the concept *LocalVariableDeclaration* and use tab or the mouse to fill in the various properties (*type, name, init*). A couple of editor customizations are necessary to make sure users can "just type" the declaration. I refer to [22] for details on how this works.

Language Modularization and Extension

I referred above to the ability to modularize and compose languages as a way of breaking down monolithic languages into manageable modules that can be combined as needed. I also already alluded to the relationship between object oriented programming and language definition in MPS. This analogy also holds for language extension and specialization. Concepts can extend other concepts, and subconcepts can be used polymorphically. Languages can extend other languages, too, and the sublanguage can contain subconcepts of concepts in the base language or can override the translation rules (generators) of concepts defined in the base language. Concept interfaces are also available. Using the *Adapter* pattern [29], unrelated concepts can be made to fit together. To use a *B* in places where an *A* (or subtypes) is expected, an adapter *BAdapter* that extends *A* and contains or references a *B* is necessary. As shown in [22], this approach supports embedding of completely unrelated languages.

Languages also define translation rules to lower-level languages or to text. MPS includes an incremental translation engine that reduces program code as long as translation rules are available for the program elements. At the end, text generators output regular program text that can be fed into a compiler.

The language modularization and composition features are extremely useful for handling customization variability, because DSLs that describe a specific aspect of the overall system can be easily integrated with the languages used to implement the rest of the system.

3. Language Annotations

With MPS it is possible to add additional data to program elements that has not been "planned for" when designing the original language. It is possible for language *X* to contribute properties to elements of language *Y* without invasively changing language *Y*. This is a little bit like introductions in aspect oriented programming [2]. It is an extremely interesting feature for adding feature dependencies, i.e. as a way of implement configuration variability. Section 4 shows how this feature is used in the context of embedded systems development.

Defining an annotation for feature dependencies

As figure 2 shows, an annotation declares which elements it targets (*A* in the example). As a consequence, this element (and all its subtypes) appears to possess the additional child *r* declared by the annotation. The child can be used like any other child of *A*. The annotation can reside in a different language than the target elements, supporting external, a-posteriori non-invasive extension of languages.

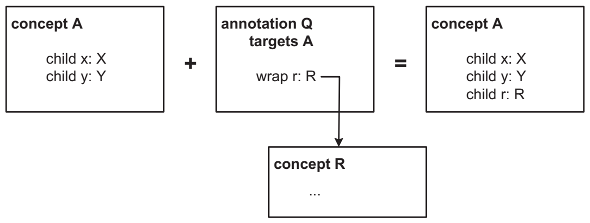
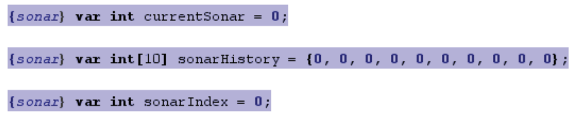


Figure 2. Annotations can add properties to concepts without invasive modification

It is, however, not enough to add the additional property to the program elements. It also has to be rendered in the editor, so the editor needs to be adapted as well. For example, if a program element has a feature dependency annotation, the actual feature expression must be shown near the element. Figure 3 shows an example of three variables that are annotated with a dependency to the *sonar* feature.



```
{sonar} var int currentSonar = 0;

{sonar} var int[10] sonarHistory = {0, 0, 0, 0, 0, 0, 0, 0, 0, 0};

{sonar} var int sonarIndex = 0;
```

Figure 3. Feature dependencies annotated to variable declarations

This is a placeholder cell which, at runtime, is replaced with the editor of the node to which the annotation has been added. So, in essence, it means that the editor of the annotation element *R* wraps around and embeds the editor to which the annotation has been added.

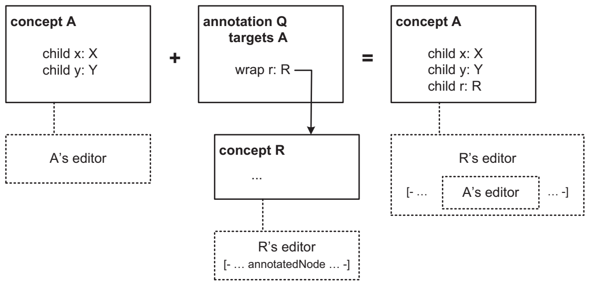
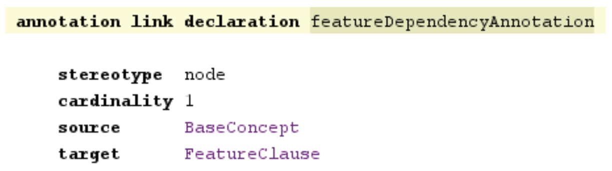


Figure 4. The editor of the annotation "wraps around" the editor of the annotated element

Figure 4 shows how editor annotation works in principle. In the definition of the editor for the element that is "added" to the target, you can use a special editor cell type (*annotatedNode*).

Returning to the example for feature dependencies, figure 5 is the definition of the annotation. It contributes a child named *featureDependencyAnnotation* of type *FeatureClause* to *BaseConcept* and its subtypes (all language element extend *BaseConcept*, at least indirectly. It is like *java.lang.Object*).

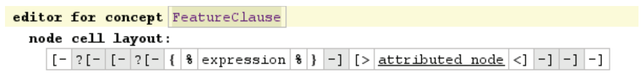


```
annotation link declaration featureDependencyAnnotation

    stereotype  node
    cardinality 1
    source      BaseConcept
    target      FeatureClause
```

Figure 5. Definition of an Annotation that adds *FeatureClause* instances to instances of any subtype of *BaseConcept*

In terms of the editor, this means that the editor of *FeatureClause* has to "wrap around" the editor of whichever other element it is contributed to. Figure 6 shows the definition of the editor for *FeatureClause:* it first includes the feature expression (such as *car && !pedestrian*) and then delegates to the node to which it has been annotated using the *attributedNode* element.

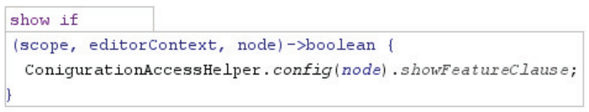


```
editor for concept FeatureClause
  node cell layout:
    [- ?[- [- ?[- { % expression % } -] [> attributed node <] -] -] -]
```

Figure 6. Definition of the annotation's editor. Note the delegation to the editor of the annotated node

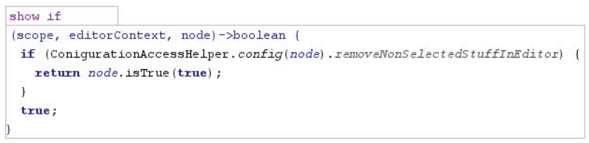
Evaluating the feature expression during projection

With the facilities described above it is possible to annotate arbitrary program elements with feature clauses. These contain a boolean expression over a set of features. Because the language that is used to define the feature expression is formally defined, it is possible to evaluate the expression in the IDE and show the program in a variant-specific way. This is done by not projecting those elements whose feature clause evaluates to false considering the current feature selection. The projection rules in the editor definition contain the respective *if* statements. Conditional projection is supported by the cells with a leading question mark (see figure 6). These are conditional cells, i.e. they are only shown in the editor at runtime if their associated condition is true. For the feature-aware code editor, the feature annotations themselves (i.e. the *{car && !pedestrian}*) are only shown if a global configuration flag *Show Feature Annotations* is true. So the condition in the conditional cell directly before the *{expression}* in the editor definition in figure 6 contains the following expression shown in figure 7:



```
show if
(scope, editorContext, node)->boolean {
  ConigurationAccessHelper.config(node).showFeatureClause;
}
```

Figure 7. Expression that makes sure the feature clause is only shown if the configuration allows it



```
show if
(scope, editorContext, node)->boolean {
  if (ConigurationAccessHelper.config(node).removeNonSelectedStuffInEditor) {
    return node.isTrue(true);
  }
  true;
}
```

Figure 8. Expression that makes sure the whole annotated element is only shown if the configuration allows it

Also, in the projection mode that shows a program variant, we want to make sure that the program element (e.g. a procedure, or an *if* statement) to which a feature expression is annotated is not shown, if the feature expression is *false*. This is achieved by the outer conditional cell in figure 6 that surrounds the expression and the attributed node. The condition is shown in figure 8.

4. The Proof of Concept

Together with Bernhard Merkle, the author is currently working on a modular language for embedded development based on C (MEL - Modular Embedded Language). It is described in detail in another paper [3]. Among other things it contains support for product line variability as described in the previous section. This section is a brief overview of the language.

Embedded Software Development Language

Embedded systems are becoming more and more software intensive. Consequently, software development plays an increasingly important part in embedded system development, and the software becomes bigger and more complex. Traditional embedded system development approaches use a variety of tools for various aspects of the system, making tool integration a major headache. Some of the specific problems of embedded software development include the limited capability for meaningful abstraction in C, some of C's "dangerous" features (leading to various coding conventions such as Misra-C [4]), the proprietary and closed nature of modeling tools, the integration of models and code, traceability to requirements [5], long build times as well as the consistent implementation of product line variability [6].

To address these issues, we propose a modular modeling and programming language and IDE that supports higher-level abstractions and system-specific extensions based on a projectional language workbench and to use code generation to C as a way of integrating with existing compilers and platforms. The proposed language uses C as its core and adds several useful extensions, including a module system with visibility rules, physical quantities (as opposed to just *ints* and *floats*), first-class state machines, dataflow ports, mathematical notations, memory mapping and bit fields, as well as first-class support for various inter-process communication approaches (shared memory, message passing, bus communication).

As a proof of concept, we are currently building a first cut of this modular embedded language (MEL) based on JetBrains MPS. We use Lego Mindstorms [7] as the target platform together with the OSEK [30, 8] operating system. C and OSEK are widely used in automotive systems, so the technologies used in the prototype are relevant in real systems. The current baseline showcase is a simple line follower robot. It uses a single light sensor to follow a thick black line. It keeps track of the curving line by changing the speed of motors that drive the two wheels. The current state of the prototype contains language modules for components, tasks, state machines, bit-level data structures, physical quantities, documentation annotations, basically all of C as well as support for product line variability and requirements traces.

Configuration Variability - Feature Annotations

Lego is a good way of showing product line variability because it is easy to clip on variant specific hardware. The following two optional hardware elements are available:

- a bumper at the front of the robot that stops it if the bumper is pressed. Essentially, this is a collision sensor.
- a sonar sensor, that temporarily stops the robot if something steps into its way. This is a collision prevention system.

In Figure 9, configuration A shows the robot in its basic setup (*bumper = false* and *sonar = false*), B shows the configuration with the bumper and C shows bumper and sonar.

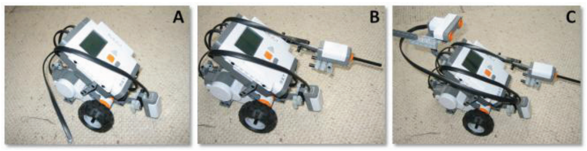
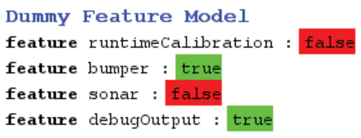


Figure 9. Three different variants of the robot

Feature dependencies are a way to implement configuration variability in programs. Handling configuration variability requires two ingredients. First, a set of variation points (aka features in the feature modeling approach [11]) and the constraints among them have to be defined. Second, program elements have to be annotated with expressions over these features. These expressions determine whether a piece of program is in a variant or not.

In a real-life system, feature management happens in an external tool (such as pure::variants [12]). In the proof-of-concept, just like with the requirements, features are managed in a simple list. Features can be selected (see figure 9) to determine whether they are in the system or not (which will become relevant later).

The second ingredient are feature annotations, i.e. annotations on program elements that contain an expression that determines which features it depends on, and how. As described above, programs can be projected in a way that shows the feature annotations directly on the element it is attached to (Figure 10). Although the figure contains only dependencies on single features, we can use boolean expressions in the feature annotation such as *{bumper && sonar && !debugOutput}*. This is actually a small sublanguage for boolean expressions (again with code completion into the feature model, error checking etc.).

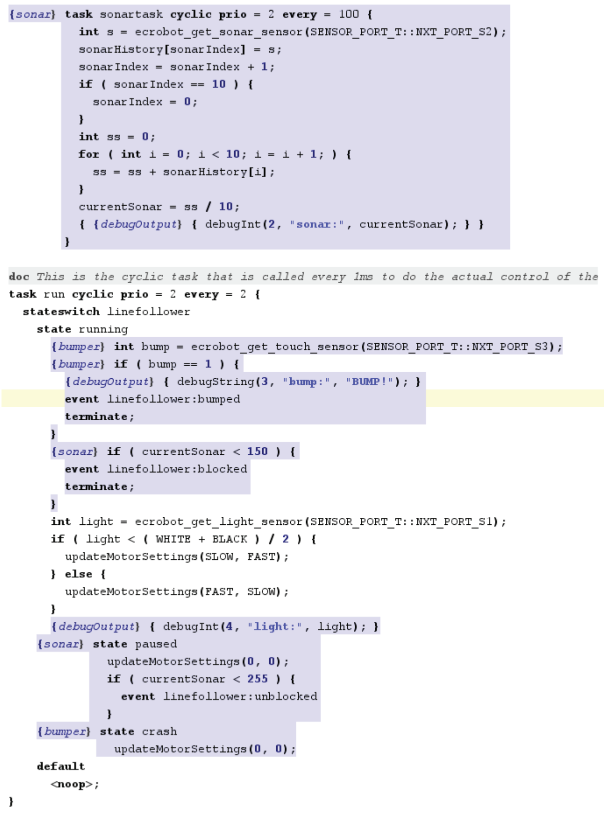


```
Dummy Feature Model
feature runtimeCalibration : false
feature bumper : true
feature sonar : false
feature debugOutput : true
```

Figure 9. The dummy feature model for the line follower robot (also contains *true/false* switches to define a variant)

As can be seen from Figure 10, feature dependency expressions can be annotated to any program element. Annotated elements have a grey background and the feature annotation expression is given on the left of the element. Alternatively it would also be possible to assign a specific color to each feature and then use the respective color as the background for the elements annotated with this feature (as done by CIDE [19]).

By flipping a switch in the overall projection settings it is possible to show the program in a variant-specific way. For example, if we switch off the *debugOutput* and the *bumper* feature, the code in Figure 11 will result.



```
{sonar} task sonartask cyclic prio = 2 every = 100 {
          int s = ecrobot_get_sonar_sensor(SENSOR_PORT_T::NXT_PORT_S2);
          sonarHistory[sonarIndex] = s;
          sonarIndex = sonarIndex + 1;
          if ( sonarIndex == 10 ) {
            sonarIndex = 0;
          }
          int ss = 0;
          for ( int i = 0; i < 10; i = i + 1; ) {
            ss = ss + sonarHistory[i];
          }
          currentSonar = ss / 10;
          { {debugOutput} { debugInt(2, "sonar:", currentSonar); } }
        }

doc This is the cyclic task that is called every 1ms to do the actual control of the
task run cyclic prio = 2 every = 2 {
  stateswitch linefollower
    state running
      {bumper} int bump = ecrobot_get_touch_sensor(SENSOR_PORT_T::NXT_PORT_S3);
      {bumper} if ( bump == 1 ) {
        {debugOutput} { debugString(3, "bump:", "BUMP!"); }
        event linefollower:bumped
        terminate;
      }
      {sonar} if ( currentSonar < 150 ) {
        event linefollower:blocked
        terminate;
      }
      int light = ecrobot_get_light_sensor(SENSOR_PORT_T::NXT_PORT_S1);
      if ( light < ( WHITE + BLACK ) / 2 ) {
        updateMotorSettings(SLOW, FAST);
      } else {
        updateMotorSettings(FAST, SLOW);
      }
      {debugOutput} { debugInt(4, "light:", light); }
    {sonar} state paused
              updateMotorSettings(0, 0);
              if ( currentSonar < 255 ) {
                event linefollower:unblocked
              }
    {bumper} state crash
               updateMotorSettings(0, 0);
    default
      <noop>;
}
```

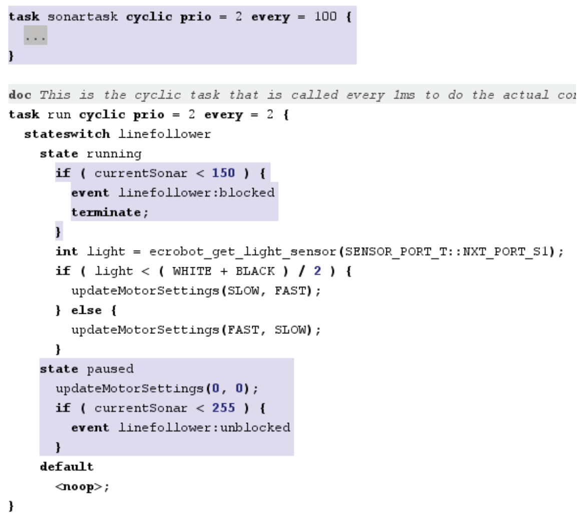
Figure 10. A piece of the overall linefollower program projected with feature annotations

Note that these projections still contain the grey highlight for parts that are feature dependent. This can be turned off, too. Also, the feature dependency expressions can be shown in this view if requested. It is important to point out that this is not a read-only projection! Rather, the program can still be edited while shown in the variant-specific way.

As mentioned above, for compilation the program is generated into plain C and then compiled with the existing legacy compiler infrastructure. When generating C, the current feature configuration is taken into account. A simple transformation script is run as part of the incremental reduction process that removes all program elements whose feature clause evaluates to false, making sure they are not part of the resulting C text.

Static Validation of Feature Dependencies

Making parts of models or code optional runs the risk of producing structurally or semantically broken programs after "cutting away" all the stuff that is not configured to be in a certain variant. Detecting semantic errors in turing-complete programs is impossible in general, of course. But static correctness can be verified.



```
task sonartask cyclic prio = 2 every = 100 {
  ...
}

doc This is the cyclic task that is called every 1ms to do the actual co
task run cyclic prio = 2 every = 2 {
  stateswitch linefollower
    state running
      if ( currentSonar < 150 ) {
        event linefollower:blocked
        terminate;
      }
      int light = ecrobot_get_light_sensor(SENSOR_PORT_T::NXT_PORT_S1);
      if ( light < ( WHITE + BLACK ) / 2 ) {
        updateMotorSettings(SLOW, FAST);
      } else {
        updateMotorSettings(FAST, SLOW);
      }
    state paused
      updateMotorSettings(0, 0);
      if ( currentSonar < 255 ) {
        event linefollower:unblocked
      }
    default
      <noop>;
}
```

Figure 11. A part of the program with *debugOutput* and *bumper* switched off (pls compare with Figure 10)

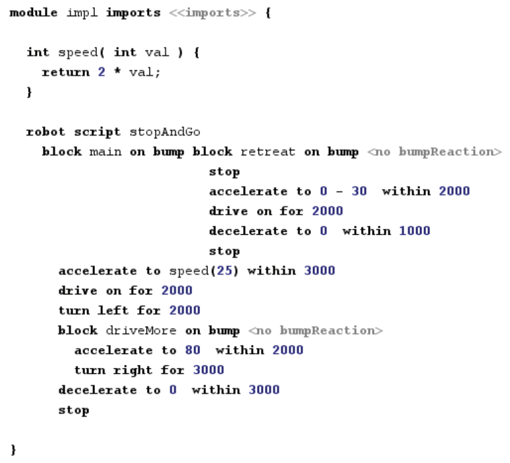
Consider that in MPS (and in projectional editors in general) every element is a node with a unique identity. Relationships between elements are expressed with actual references to these unique identities. A structurally broken program is one where a referencing element is in the code for a given variant, but the reference target is not. Static validation of feature dependencies requires showing that for any (valid) feature combination, no such "dangling pointer" will result. The following is a simple approach to verify this:

- Calculate all combinations of all features (i.e. all variants)
- For each referencing element R, collect all feature combinations C_R for which this element will be in the variant code
- For each reference target element T of R, collect all combinations C_T for which this element will be in the variant code
- If C_R is not a subset of C_T, an error has been detected

This algorithm has been implemented in the prototype and it works well in principle. The fact that all references can be followed easily, and the fact that feature dependencies are expressed as expressions based on a formal expression language makes implementing this algorithm simple - and it does work for small sets of configuration features. But of course the set of possible feature combinations grows exponentially over the number of features, so for real-world sized systems it will not work. The following steps could be taken to address this:

- In real systems, the set of features is not unrestricted, they have constraints among each other. This limits the size of the set of valid feature combinations (i.e. variants)
- Feature macros can be introduced, i.e. features that encapsulate a set of other features and their constraints (e.g. *fast := !small && !dynamic*). If feature dependencies refer to the macro features, they can be seen as *one* combination and the combinatorics behind them can be ignored.
- If only a part of a program needs to be validated, only the feature combinations involving the features referenced from the respective part of the program need to be calculated.
- Finally, using a solver instead of the try-all-combinations brute force approach may yield even more scalable results.

We will explore these alternatives as part of our in the future work (see below).



```
module impl imports <<imports>> {

  int speed( int val ) {
    return 2 * val;
  }

  robot script stopAndGo
    block main on bump block retreat on bump <no bumpReaction>
                        stop
                        accelerate to 0 - 30  within 2000
                        drive on for 2000
                        decelerate to 0  within 1000
                        stop
      accelerate to speed(25) within 3000
      drive on for 2000
      turn left for 2000
      block driveMore on bump <no bumpReaction>
        accelerate to 80  within 2000
        turn right for 3000
      decelerate to 0  within 3000
      stop

}
```

Figure 12. A simple robot routing script

Customization Variability - a DSL on top

Consider now a robot vendor who sells Lego robots with two wheels that can follow a predefined route. Of course, each customer wants a different predefined route. The vendor has to develop a different route-driving program for each customer. Of course this can be achieved with tasks, state machines, variables and procedures, or in other words, the general-purpose MEL. But it would be better if a domain specific language for defining routes was available. In PLE terminology, the DSL would be used to express the problem domain and a transformation would map this to a solution domain implementation.

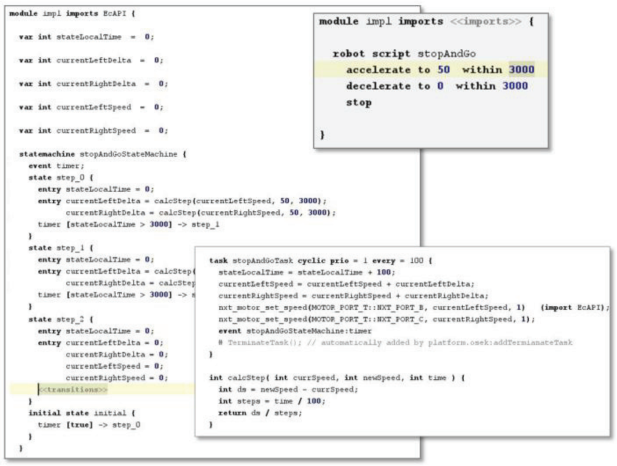
The program in figure 12 is an example expressed with such as route definition DSL, it uses native route-definition constructs. Since the robot routing language extends the core language, it can be embedded in a module - like the general purpose MEL constructs. The robot script can even call procedures. The robot routing language is executed by transformation into the following MEL constructs:

- a state machine that keeps track of the current command/step
- module variables that remember the current speeds for the two motors
- module variables that store the current deltas for the speeds of both motors to be able to "ramp up" the speeds in accelerate and decelerate commands
- a cyclic tasks that updates motor speeds based on the currently selected deltas.

Figure 13 shows a robot script together with the lower-level program that results from the transformation.

Combining the DSL and Feature Annotations

It useful to combine customization and confguration variability. In the example this would mean that we can attach feature expressions to robot script programs. This is of course also possible. The feature annotations are completely generic and make no assumption about the language to which they are attached. Consequently, they can be used with the robot DSL in the same way as with the lower level programs.



```
module impl imports EcAPI {

  var int stateLocalTime  =  0;

  var int currentLeftDelta  =  0;

  var int currentRightDelta  =  0;

  var int currentLeftSpeed  =  0;

  var int currentRightSpeed  =  0;

  statemachine stopAndGoStateMachine {
    event timer;
    state step_0 {
      entry stateLocalTime = 0;
      entry currentLeftDelta = calcStep(currentLeftSpeed, 50, 3000);
            currentRightDelta = calcStep(currentRightSpeed, 50, 3000);
      timer [stateLocalTime > 3000] -> step_1
    }
    state step_1 {
      entry stateLocalTime = 0;
      entry currentLeftDelta = calcStep(
            currentRightDelta = calcStep
      timer [stateLocalTime > 3000] -> s
    }
    state step_2 {
      entry stateLocalTime = 0;
      entry currentLeftDelta = 0;
            currentRightDelta = 0;
            currentLeftSpeed = 0;
            currentRightSpeed = 0;
      <<transitions>>
    }
    initial state initial {
      timer [true] -> step_0
    }
}
```

```
module impl imports <<imports>> {

  robot script stopAndGo
    accelerate to 50  within 3000
    decelerate to 0  within 3000
    stop

}
```

```
task stopAndGoTask cyclic prio = 1 every = 100 {
  stateLocalTime = stateLocalTime + 100;
  currentLeftSpeed = currentLeftSpeed + currentLeftDelta;
  currentRightSpeed = currentRightSpeed + currentRightDelta;
  nxt_motor_set_speed(MOTOR_PORT_T::NXT_PORT_B, currentLeftSpeed, 1)   (import EcAPI);
  nxt_motor_set_speed(MOTOR_PORT_T::NXT_PORT_C, currentRightSpeed, 1);
  event stopAndGoStateMachine:timer
  # TerminateTask(); // automatically added by platform.osek:addTermianateTask
}

int calcStep( int currSpeed, int newSpeed, int time ) {
  int ds = newSpeed - currSpeed;
  int steps = time / 100;
  return ds / steps;
}
```

Figure 13. A simple robot script (top right, grey) and the lower level MEL program it is transformed into

5. Future Work

Future work will progress in three main directions: additional language concepts, real world-validation and integration of existing feature modeling tools.

Feature Modeling Tools

Currently we use a flat list of features (each basically boolean switches) as our feature model. We chose this approach because it is trivial to implement, and the point we wanted to make with our work was not to implement a new feature modeling tool.

However, to make our approach more useful in practice, integration with tools such as pure-variants [12]. We will simply import the list of features as well as the constraints among them. This will allow us to refer to these features from within feature expressions, and it will allow us to exploit the constraints between the features when we calculate whether a program is structurally valid.

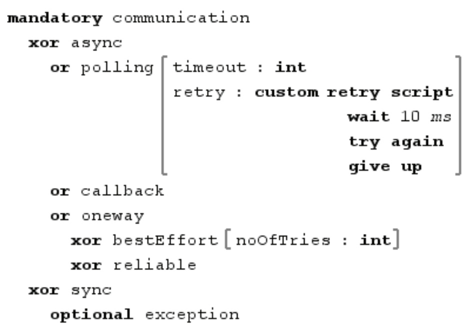
Additional Language Concepts

An alternative to overlaying configuration over program or model code is to make the feature model the main configuration tool and add DSL code to it. In most feature management tools (for example, pure::variants [12]) features can have parameters. For example, when selecting a *buffered* feature for a communication protocol product line, a *buffer size* parameter can be specified. Generalizing this approach leads to the following:

- each feature may define any number of parameters. These cannot just be simple types (*int*, *string*, *boolean*) but can include DSLs.
- When a feature is selected, a value for the parameter that complies to the parameter's type has to be supplied. For DSL-typed parameters, this means that a model that conforms to the DSL must be supplied.

Because projectional language workbenches can integrate models using any combination of DSLs, this approach is feasible. Figure 14 shows a very early prototype of this; a retry algorithm is used as the value for the *retry* parameter of the *polling* feature.

Optionally taking away program elements if they are not included in the variant is only one way of implementing variability. The approach is often called negative variability. The other alternative is to conditionally add to a minimal core - positive variability. The advantage is that the minimal core remains small, quite in contrast to negative variability where the overall program that includes all variants can grow quite large. Like in AOP [2], positive variability requires pointcuts to define where to add the additional program elements to the core. Future work will focus on positive variability MEL as a means of implementing variability.



```
mandatory communication
  xor async
    or polling [ timeout : int
               [ retry : custom retry script
               [                 wait 10 ms
               [                 try again
               [                 give up
    or callback
    or oneway
      xor bestEffort [ noOfTries : int ]
      xor reliable
  xor sync
    optional exception
```

Figure 14. A DSL snippet in a feature model

As mentioned above, the current approach to feature validation is brute force and does not at all scale. One aspect of our future work will address this issue. We've already started collaborating with a university who has experience in this regard.

Real-World validation

The other main avenue of future work is real world validation. We are currently in the process of starting up a project to do a real prototype - something more realistic than the Lego Mindstorms example we are currently building. The connection to real requirements management systems and to variant management tools will be a part of this prototype.

6. Related Work

The idea of using DSLs to describe variability in product lines is not new. Various authors have published about this [13,14,15] and the approach is used in practice. The approach described in this paper is different since the various DSLs can be mixed and integrated. Language composition for textual languages is not easily possible with non-projectional editors, although progress is being made, as exemplified by [18, 28].

Overlaying configuration variability over customization variability has been done before, too. The C preprocessor can be used to this effect using *#ifdefs*. The approach can also be used on models. For example, Krzysztof Czarnecki and his group have overlaid feature-based variability over UML diagrams [16].The approach described in this paper is different in that configurative variability can be overlaid over models and code in the same way - there is no difference between the two in the first place. Since the feature expressions are also a formal language, the expressions can be formally checked and interpreted. The ability to show the program/model code with feature clauses enabled or not, and to show the (and edit!) the model in a variant-specific way is also radically different from these tools. CIDE, a specific solution for C code is described in [19]. However, the approach described in this paper is different since it works for any language within MPS. Also, the approach described in this paper supports the combination of the annotation-based approach with language composition and DSLs. VML [17] is another tool (based on Eclipse EMF) that can map configurative variability to arbitrary models. However, since source code (C, Java) is not represented with EMF in Eclipse, a special solution had to be created to "adapt" VML to source code.

Showing statically that every valid variant of the feature model will result in a structurally valid program has been done before by [27] for the case of UML models and OCL constraints. Also the tool developed by Czarnecki et al. [16] has static validation to make sure that every variant of the UML model is structurally correct. Another approach for the same problem is described as part of the AHEAD methodology in [20]. Verifying that only "correct" programs are synthesized by program synthesis is a popular resarch topic [25, 26, 27,31]. We will use the approaches described in these papers in our future work, since the focus of our work is not primarily on this kind of verification.

Krzysztof Czarnecki and his group are currently working on a very interesing language: Clafer [24], a combination of structural class modeling and feature modeling. Krzysztof and his group are planning to integrate Clafer into the MPS prototype. One area where the approach described in this paper is more flexible is that we can use arbitrary DSLs and syntax to describe structural variability, whereas Clafer is essentially limited to class (or meta-) modeling.

7. Evaluation & Conclusion

As we continue to build Mindstorms applications with our language, it turned out that it is useful to extend plain C with embedded-specific concepts. Programs can be read and analyzed more easily: the more the language constructs resemble the intent of the programmer, the more meaningful analyses can be.

It is feasible to package the various aspects into separate language modules and make incremental extension possible. It is also surprisingly little effort to build language extensions: developing the basic C implementation has taken us about 3 weeks. Adding the statemachine facilities has been done in one afternoon. Creating the *robot routing* DSL on top was a matter of 4 hours, including the mapping down to tasks and state machines. Consequently, the concept of building DSLs to express some aspect of a product line is absolutely feasible.

Adding variability annotations to program elements is not fundamentally new. *#ifdefs* in C programs can be used for a similar approach. However, the ability to reliably evaluate the expressions, show and edit the programs in variant-specific ways as well as the static validation of feature dependencies has proven useful even in our simple examples.

Acknowledgments

My thanks go to Konstantin Solomatov of JetBrains who tirelessly supported my efforts of learning MPS, as well as to Christoph Elsner for his feedback on this paper.

References

[1] Fowler, M., *Language Workbenches: The Killer-App for Domain Specific Languages?*, http://martinfowler.com/articles/languageWorkbench.html

[2] Kiczales, G., Lamping, J., Mendhekar, A., Maeda, C., Lopes, C., Loingtier, J., Irwin, J. *Aspect-Oriented Programming*. Proceed ings of ECOOP 1997, vol.1241. pp. 220–242.

[3] Markus Voelter, *Embedded Software Development with Projectional Language Workbenches*, Proc. of MODELS 2010,

[4] MISRA Group, *Misra-C*, http://www.misra-c2.com/

[5] Gotel, O., Finkelstein, A., *An Analysis of the Requirements Traceability Problem*, Proc. of First International Conference on Requirements Engineering, 1994, pages 94-101

[6] Software Engineering Institute, *Software Product Lines*, http://www.sei.cmu.edu/productlines/

[7] Lego SA, *Mindstorms*, http://mindstorms.lego.com

[8] Sourceforge.net, *nxtOSEK*, http://lejos-OSEK.sourceforge.net/ Osek

[9] IBM Corp, *Requisite Pro - a Requirements Management Tool*, http://www-01.ibm.com/software/awdtools/reqpro/

[10] IBM Corp, *Rational DOORS*, http://www-01.ibm.com/software/awdtools/doors/productline/

[11] Kang, K.C. and Cohen, S.G. and Hess, J.A. and Novak, W.E. and Peterson, A.S., *Feature-oriented domain analysis (FODA) feasibili ty study*, Technical Report CMU/SEI-90-TR-021, SEI, Carnegie Mellon University, November 1990

[12] Pure Systems GmbH, *pure::variants*, http://www.pure-systems.com/pure_variants.49.0.html

[13] Batory, D., Johnson, C., MacDonald, B., von Heeder, D., *Achieving Extensibility through Product-Lines and Domain-Specific Lan guages: A Case Study*, LNCS, Volume 1844/2000

[14] Mernik, M., Heering, J., Sloane, A., *When and how to develop domain-specific languages*, ACM Computing Surveys (CSUR), Volume 37 , Issue 4

[15] Tolvanen, J., Kelly, S., *Defining Domain-Specific Modeling Languages to Automate Product Derivation: Collected Experiences*, Lecture Notes in Computer Science, Volume 3714/2005

[16] Czarnecki, K., Antkiewicz, M.. *Mapping Features to Models: A Template Approach Based on Superimposed Variants*. In Proceed ings of GPCE'05, 2005

[17] Loughran, N., Sanchez, P., Garcia, A., Fuentes, L., *Language Support for Managing Variability in Architectural Models*, Lecture Notes in Computer Science, Volume 4954/2008

[18] Bravenboer, M., Visser, E., *Designing Syntax Embeddings and Assimilations for Language Libraries*, ATEM'07 and http://swerl.tudelft.nl/bin/view/EelcoVisser

[19] Kästner, C., *CIDE: Virtual Separation of Concerns*, http://wwwiti.cs.uni-magdeburg.de/~ckaestne/

[20] Thaker, S., Batory, D., Kitchin D., Cook, W., *Safe Composition of Product Lines*, GPCE 2007, http://userweb.cs.utexas.edu/~wcook/papers/gpce07/ThakerGPCE07.pdf

[21] JetBrains Inc, Meta Programming System (MPS), http://jetbrains.com/mps

[22] Voelter, M., Solomatov, K., *Language Modularization and Composition with Projectional Language Workbenches illustrated with MPS*, submitted to SLE 2010

[23] Wikipedia, View Model, http://en.wikipedia.org/wiki/View_model

[24] Krzyzsztof Czarnecki, *Feature and Class Models in Clafer: Mixed, Specialized, and Coupled*, personal communication, now probably available at http://gsd.uwaterloo.ca/~kczarnec/

[25] Huang S., Zook D., Smaragdakis, Y., *Statically Safe Program Generation with SafeGen*, GPCE 2005

[26] Krishnamurthi S., Fisler K.,, Greenberg M.. *Verifying Aspect Advice Modularly*, ACM SIGSOFT 2004

[27] Czarnecki, K., Pietroszek, K.. *Verification of Feature-Based Model Templates Against Well-Formedness OCL Constraints*. GPCE 2006

[28] Bravenboer, M., Visser, E., *Parse Table Composition, Separate Compilation and Binary Extensibility of Grammars,* SLE'08 and http://swerl.tudelft.nl/bin/view/EelcoVisser

[29] Gamma, E., Helm, R., Johnson R., Vlissides, J., *Design Patterns,* Addison-Wesley, 1994

[30] Continental Automotive, *http://osek-vdx.org/*

[31] Kästner, C., Apel, S., Trujillo, S., Kuhlemann M., Batory, D., *Guaranteeing Syntactic Correctness for all Product Line Variants: A Language-Independent Approach*, TOOLS Europe 2009

Pure Delta-oriented Programming [*]

Ina Schaefer[1] [†] Ferruccio Damiani[2]

[1]Chalmers University of Technology, 421 96 Gothenburg, Sweden
[2]Dipartimento di Informatica, Università di Torino, C.so Svizzera, 185 - 10149 Torino, Italy

Abstract

Delta-oriented programming (DOP) is a modular approach for implementing software product lines. Delta modules generalize feature modules by allowing removal of functionality. However, DOP requires to select one particular product as core product from which all products are generated. In this paper, we propose *pure delta-oriented programming* (Pure DOP) that is a conceptual simplification of traditional DOP. In Pure DOP, the requirement of one designated core product is dropped. Instead, program generation only relies on delta modules comprising program modifications such that Pure DOP is more flexible than traditional DOP. Furthermore, we show that Pure DOP is a true generalization of FOP and supports proactive, reactive and extractive product line engineering.

Categories and Subject Descriptors D.1.5 [*Programming Techniques*]: Object-oriented Programming; D.3.3 [*Programming Languages*]: Language Constructs and Features

General Terms Design, Languages, Theory

Keywords Software Product Line, Feature-oriented Programming, Delta-oriented Programming, Program Generation

1. Introduction

A *software product line* (SPL) is a set of software systems with well-defined commonalities and variabilities [12, 27]. The approaches to implementing SPL in the object-oriented paradigm can be classified into two main directions [19]. First, annotative approaches (e.g., [4, 17]) mark the source code of all products with respect to product features and remove marked code for particular feature configurations. Second, *compositional approaches* [23], associate code fragments to product features that are assembled to implement a given feature configuration.

Feature-oriented programming (FOP) [7] is a prominent approach for implementing SPLs by composition of *feature modules*. A feature module directly corresponds to a product feature. In the context of object-oriented programming, feature modules can introduce new classes or refine existing ones by adding fields and methods or by overriding existing methods. In *delta-oriented programming* (DOP) [29], feature modules are generalized to delta modules that additionally allow the removal of classes, fields and methods and that can refer to any combination of features. DOP requires selecting one particular product as designated core product. The core product is implemented in the core module. From this core module, all other products are generated by delta module application. However, the requirement of the core product makes it difficult to deal with product line evolution, for instance, if the product line evolves such that the original core product is no longer a valid product. Furthermore, the uniquely determined core product prevents a true generalization of FOP by DOP, since feature module composition in FOP may start from several different base feature modules that may not correspond to valid products.

In this paper, we propose *pure delta-oriented programming* (Pure DOP) as a conceptual simplification of traditional DOP [29], which we will call Core DOP in the following. In Pure DOP, the requirement to chose one product as core product is dropped. Instead, only delta modules are used for product generation. Thus, we call the approach *Pure DOP*. A delta module can specify additions, removals classes or modifications of classes. In order to define a product line over a set of delta modules, each delta module is attached an application condition determining for which feature configurations the modifications of the delta module have to be applied. This creates the connection between the modifications of the delta modules and the product features [16]. Additionally, the delta modules can be partially ordered to ensure that for every feature configuration a uniquely defined product is generated.

The contribution of this work is twofold. First, Pure DOP relaxes the requirement of a single valid core product. This makes Pure DOP more flexible than Core DOP [29]. Pure DOP is a true generalization of FOP since every FOP product line can be understood as a Pure DOP product line which is not obvious for Core DOP. Further, Pure DOP supports proactive, reactive and extractive product line development [22] by allowing program generation from any set of existing legacy product implementations which is not directly possible with Core DOP. Second, in the presentation of (Pure) DOP given in this paper, the application conditions for delta modules, as well as the delta module ordering, are only defined when a product line is specified. In contrast, in the traditional presentation of (Core) DOP [29], application conditions and ordering are fixed for each delta module. The separation of application conditions and application ordering from the specification of the modifications in a delta module increases the reusability of delta modules and allows developing different product lines over the same set of delta modules.

The paper is organized as follows: In Section 2, we present Pure DOP of JAVA programs and show its formalization LPΔJ using LJ (LIGHTWEIGHT JAVA) [32] as base language for the generated products in Section 3. We show that Pure DOP is a

[*] Work partially supported by the German-Italian University Centre (Vigoni program) and by MIUR (PRIN 2009 DISCO).

[†] This author has been supported by the Deutsche Forschungsgemeinschaft (DFG) and by the EU project FP7-ICT-2007-3 HATS.

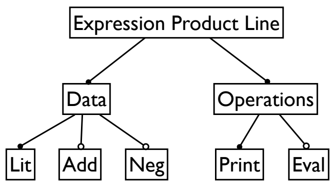


Figure 1. Feature model for Expression Product Line

true generalization of FOP by providing an embedding of LFJ (LIGHTWEIGHT FEATURE JAVA) [13] into LPΔJ in Section 4. We demonstrate that Pure DOP supports proactive, extractive and reactive SPLE in Section 5. We show that Pure DOP is a conceptual simplification of Core DOP in Section 6.

2. Pure Delta-oriented Programming

In order to illustrate the main concepts of Pure DOP, we use the *expression product line* (EPL) as described in [23]. The EPL is based on the *expression problem* [35], an extensibility problem, that has been proposed as a benchmark for data abstractions capable to support new data representations and operations. We consider the following grammar:

```
Exp ::= Lit | Add | Neg
Lit ::= <non−negative integers>
Add ::= Exp "+" Exp
Neg ::= "-" Exp
```

Two different operations can be performed on the expressions described by this grammar: printing, which returns the expression as a string, and evaluation, which computes the value of the expression. The products in the EPL can be described by two feature sets, the ones concerned with data Lit, Add, Neg and the ones concerned with operations Print and Eval. Lit and Print are mandatory features. The features Add, Neg and Eval are optional. Figure 1 shows the feature model [16] of the EPL.

Pure Delta Modules The main concept of pure DOP are delta modules which are containers of modifications to an object-oriented program. The modifications inside a delta module act on the class level by adding, removing and modifying classes. A class can be modified by changing the super class, by adding and removing fields and methods and by modifying methods. The modification of a method can either replace the method body by another implementation, or wrap the existing method using the **original** construct. The **original** construct expresses a call to the method with the same name before the modifications and is bound at the time the product is generated. Before or after the **original** construct, other statements can be introduced wrapping the existing method implementation. The **original** construct (similar to the **Super**() call in AHEAD [7]) avoids a combinatorial explosion of the number of delta modules in case the original method has to be wrapped differently for a set of optional features. Listing 1 contains the delta module for introducing the Lit feature. Listing 2 contains the delta modules for incorporating the Print and Eval features by modification of the class `Lit`.

Pure Delta-oriented Product Lines The delta-oriented specification of a product line comprises the set of product features, the set of valid feature configurations and the set of delta modules necessary to implement all valid products. Furthermore, the specification of a product line in Pure DOP associates each delta module with the set of features configurations in which the delta modules has to be applied by attaching an application condition in a **when** clause. The application condition is a propositional constraint over the set

```
delta DLit{
  adds interface Exp {
  }
  adds class Lit implements Exp {
    int value;
    Lit(int n) { value = n; }
  }
}
```

Listing 1: Delta module for Lit feature

```
delta DLitPrint{
  modifies interface Exp {
    void print();
  }
  modifies class Lit implements Exp {
   adds void print() { System.out.println(value); }
  }
}

delta DLitEval{
  modifies interface Exp {
    adds eval();
  }
  modifies class Lit {
    adds int eval() { return value; }
  }
}
```

Listing 2: Delta modules for Print and Eval features

```
delta DAdd {
  adds class Add implements Exp {
    Exp expr1;
    Exp expr2;
    Add(Exp a, Exp b) { expr1 = a; expr2 = b; }
  }
}

delta DAddPrint {
  modifies class Add {
    adds void print() { expr1.print(); System.out.print(" + "); expr2.print(); }
  }
}

delta DAddEval {
  modifies class Add {
    adds int eval() { return expr1.eval() + expr2.eval(); }
  }
}
```

Listing 3: Delta modules for Add, Print and Eval features

```
delta DNeg {
  adds class Neg implements Exp {
   Exp expr;
   Neg(Exp a) { expr1 = a; }
  }
}

delta DNegPrint {
  modifies class Neg {
    adds void print() { System.out.print("-("); expr.print(); System.out.print(")");}
  }
}

delta DNegEval{
   modifies class Neg {
     adds int eval() { return (−1) ∗ expr.eval(); }
   }
}
```

Listing 4: Delta modules for Neg, Print and Eval features

```
features Lit, Add, Neg, Print, Eval
configurations Lit & Print
deltas
  [ DLit,
    DAdd when Add,
    DNeg when Neg ]

  [ DPrint,
    DEval when Eval,
    DAddPrint when Add,
    DAddEval when (Add & Eval),
    DNegPrint when Neg,
    DNegEval when (Neg & Eval) ]
```

Listing 5: Pure DOP specification of the EPL

of features. Since only feature configurations which are valid according to the feature model are used for program generation, the application conditions attached to delta modules have to be understood as a conjunction with the formula describing the set of valid feature configurations.[1] The application condition creates the link from the features in the feature model to the delta modules. In this way, we can specify delta modules for combinations of features to solve the optional feature problem [20].

In order to obtain a product for a particular feature configuration, the modifications specified in the delta modules with valid application conditions are applied incrementally to the previously generated product. The first delta module is applied to the empty product. All other delta modules are applied to the respective intermediate product. The modifications of a delta model are applicable to a (possibly empty) product if each class to be removed or modified exists and, for every modified class, if each method or field to be removed exists, if each method to be modified exists and has the same header as the modified method, and if each class, method or field to be added does not exist. During product generation, every delta module must be applicable. Otherwise, the resulting product is undefined. In particular, the first delta module that is applied can only contain additions.

In order to ensure that each delta module is applicable during product generation, the delta modules are ordered in the specification of a pure delta-oriented product line. The order of delta module application is defined by a total order on a partition of the set of delta modules. Deltas in the same partition can be applied in any order to the previous product, but the order of the partitions is fixed. The ordering captures semantic requires relations that are necessary for the applicability of the delta modules.

Listing 5 shows a delta-oriented specification of the EPL. In this specification, application conditions are attached to the delta modules that are required to implement the different products of the EPL. The used delta modules are depicted in Listings 1, 2, 3 and 4. The order of delta module application is defined by an ordered list of the delta module sets which are enclosed by `[ .. ]`.

Product Generation The generation of a product for a given feature configuration consists of two steps, performed automatically:

1. Find all delta modules with a valid application condition; and
2. Apply the selected delta modules to the empty product in any linear ordering that is consistent with the total order on the partitioning of the delta modules.

If two delta modules add, remove or modify the same class, the ordering in which the delta modules are applied can influence the resulting product. However, for a product line implementation, it is

[1] In the examples the valid feature configurations are represented by a propositional formula over the set of features. Other representations are possible (see, e.g., [5] for a discussion of possible representations).

DMD	::=	**delta** δ $\{\overline{\text{DC}}\}$	delta module
DC	::=	**adds** cd |	delta clause
		modifies C [**extending** C] { $\overline{\text{DS}}$ } |	
		removes C	
DS	::=	**adds** fd |	delta subclause
		adds md |	
		modifies md |	
		modifies wmd |	
		removes a	
wmd	::=	ms $\{\bar{\text{s}}$; **original**(); $\bar{\text{s}}$; **return** y; $\}$	method wrapper

Figure 2. LPΔJ: syntax of delta modules

essential to guarantee that for every valid feature configuration exactly one product is generated. This property is called *unambiguity* of the product line. For unambiguity, the delta modules in each partition must be compatible. This means that if one delta module in a partition adds or removes a class, no other delta module in the same partition may add, remove or modify the same class, and the modifications of the same class in different delta modules in the same partition have to be disjoint. Defining the order of delta module application by a total ordering on a delta module partition provides an efficient way to ensure unambiguity, since only the compatibility of each partition has to be checked.

3. A Kernel Calculus for Pure Delta Modules

In this section, we introduce the syntax and the semantics of LPΔJ (LIGHTWEIGHT PURE DELTA JAVA), a kernel calculus for Pure DOP of product lines of JAVA programs. LPΔJ is based on LJ (LIGHTWEIGHT JAVA) [32]. Thus, it is particularly suitable for comparison with the formalization of FOP in LFJ (LIGHTWEIGHT FEATURE JAVA) [13].

LPΔJ *Syntax* The syntax of LPΔJ, as an extensions to LJ, is given in Figure 2. Following [15], we use the overline notation for possibly empty sequences. For instance, we write "$\bar{\text{s}}$;" as short for a possibly empty sequence of statements "$\text{s}_1; \ldots \text{s}_n$;" and "$\overline{\text{DC}}$" as short for a possibly empty sequence of delta clause definitions "$\text{DC}_1 \ldots \text{DC}_n$". Sequences of named elements (like delta clause or delta subclause definitions) are assumed to contain no duplicate names (that is, the names of the elements of the sequence must be distinct). The constructs for class definitions cd, field definitions fd, method definitions md, method signatures ms and statement s are those of LJ [32] (and of LFJ [13]). The metavariable δ ranges over delta module names.

A delta module definition DMD for a delta module with the name δ can be understood as a mapping from class names to delta clause definitions. A delta clause definition DC can specify the addition, removal or modification of a class. The adds-domain, the removes-domain and the modifies-domain of a delta module definition DMD are defined as follows:

$$\begin{array}{lcl}
\mathit{addsDom}(\text{DMD}) & = & \{\texttt{C} \mid \text{DMD}(\texttt{C}) = \textbf{adds class } \texttt{C} \cdots\} \\
\mathit{removesDom}(\text{DMD}) & = & \{\texttt{C} \mid \text{DMD}(\texttt{C}) = \textbf{removes } \texttt{C}\} \\
\mathit{modifiesDom}(\text{DMD}) & = & \{\texttt{C} \mid \text{DMD}(\texttt{C}) = \textbf{modifies } \texttt{C} \cdots\}
\end{array}$$

The modification of a class is defined by possibly changing the super class and by listing a sequence of delta subclauses DS defining modifications of methods and additions/removals of fields and methods. A delta modifies clause DC can be understood as a mapping from the keyword **extending** to an either empty or singleton set of class names and from field/method names to delta subclauses. The adds-, removes- and modifies-domain of a delta modifies-clause DC are defined as follows:

$$\begin{array}{lcl} \mathit{addsDom}(\mathrm{DC}) & = & \{\mathrm{a} \mid \mathrm{DMT}(\mathrm{a}) = \textbf{adds} \cdots \mathrm{a} \cdots\} \\ \mathit{removesDom}(\mathrm{DC}) & = & \{\mathrm{a} \mid \mathrm{DMT}(\mathrm{a}) = \textbf{removes}\ \mathrm{a}\} \\ \mathit{modifiesDom}(\mathrm{DC}) & = & \{\mathrm{m} \mid \mathrm{DMT}(\mathrm{m}) = \textbf{modifies} \cdots \mathrm{m} \cdots\} \end{array}$$

The modification of a method, defined by a delta modifies subclause, can either replace the method body by another implementation, or wrap the existing method using the **original**() call. In both cases, the modified method must have the same header as the unmodified method. The **original**() call may only occur in the body of the method provided by a delta modifies subclause **modifies** wmd. The occurrence of **original**() represents a call to the unmodified method where the formal parameters of the modified method are passed implicitly as arguments. In LFJ [13], the `Super()` construct of AHEAD [7] is modeled in the same way.

After we have defined the notion of delta modules over LJ, we can formalize LPΔJ product lines. We use the metavariables φ and ψ to range over feature names. We write $\overline{\psi}$ as short for the set $\{\overline{\psi}\}$, i.e., the feature configuration containing the features $\overline{\psi}$. A *delta module table* DMT is a mapping from delta module names to delta module definitions. A LPΔJ SPL is a 5-tuple $\mathrm{L} = (\overline{\varphi}, \Phi, \mathrm{DMT}, \Gamma, <_{\mathrm{DMT}})$ consisting of:

1. the features $\overline{\varphi}$ of the SPL,
2. the set of the valid feature configurations $\Phi \subseteq \mathscr{P}(\overline{\varphi})$,[2]
3. a delta module table DMT containing the delta modules,
4. a mapping $\Gamma : \mathit{dom}(\mathrm{DMT}) \to \Phi$ determining for which feature configurations a delta module must be applied (which is denoted by the **when** clause in the concrete examples),
5. a total order $<_{\mathrm{DMT}}$ on a partition of *dom*(DMT), called the application partial order, determining the order of delta module application.

To simplify notation, in the following we always assume a *fixed* SPL $\mathrm{L} = (\overline{\varphi}, \Phi, \mathrm{DMT}, \Gamma, <_{\mathrm{DMT}})$. We further assume that the SPL L satisfies the following sanity conditions.

(*i*) For every class name `C` (except `Object`) appearing in DMT, we have $\mathtt{C} \in (\cup_{\delta \in \mathit{dom}(\mathrm{DMT})} \mathit{addsDom}(\mathrm{DMT}(\delta)))$, meaning that every class is added at least once.

(*ii*) The mapping $\Delta : \Phi \to \mathscr{P}(\mathit{dom}(\mathrm{DMT}))$, such that $\Delta(\overline{\psi})$, the set of names of delta modules whose application condition is satisfied by the feature configuration $\overline{\psi}$, is injective and such that $(\cup_{\overline{\psi} \in \Phi} \Delta(\overline{\psi})) = \mathit{dom}(\mathrm{DMT})$, i.e., for every feature configuration a different set of delta modules is applied and every delta module is applied for at least one feature configuration.

In the following, we write $\mathit{dom}(\delta)$ as short for $\mathit{dom}(\mathrm{DMT}(\delta))$, and we write $\delta(\mathtt{C})$ as short for $\mathrm{DMT}(\delta)(\mathtt{C})$.

LPΔJ *Product Generation* A LJ program can be represented by a *class table*. A class table CT is a mapping from class names to class definitions. A delta module is *applicable* to a class table CT if each class to be removed or modified exists and, for every delta modifies clause, if each method or field to be removed exists, if each method to be modified exists and has the same header specified in method modifies subclause, and if each class, method or field to be added does not exist.

Given a delta module δ and a class table CT such that δ is applicable to CT, the application of δ to CT, denoted by $\textsc{apply}(\delta, \mathrm{CT})$, is the class table CT' defined as follows:

[2] The calculus abstracts from the concrete representation of the feature model.

$$\begin{array}{llll} \mathrm{FMD} & ::= & \textbf{feature}\ \varphi\ \{\overline{\mathrm{cd}}\ \overline{\mathrm{rcd}}\} & \text{feature module} \\ \mathrm{rcd} & ::= & \textbf{refines class}\ \mathrm{C}\ \textbf{extending}\ \mathrm{C}\ \{\ \overline{\mathrm{fd}};\ \overline{\mathrm{md}}\ \overline{\mathrm{rmd}}\ \} & \text{class refinement} \\ \mathrm{rmd} & ::= & \textbf{refines}\ \mathrm{ms}\ \{\overline{\mathrm{s}};\ \textbf{Super}();\ \overline{\mathrm{s}};\ \textbf{return}\ \mathrm{y};\} & \text{method refinement} \end{array}$$

Figure 3. LFJ: syntax of feature modules

$$\mathrm{CT}'(\mathrm{C}) = \begin{cases} \mathrm{CT}(\mathrm{C}) & \text{if } \mathrm{C} \notin \mathit{dom}(\mathrm{DMT}(\delta)) \\ \mathrm{CD} & \text{if } \delta(\mathrm{C}) = \textbf{adds}\ \mathrm{CD} \\ \textsc{apply}(\delta(\mathtt{C}), \mathrm{CT}(\mathtt{C})) & \text{if } \mathtt{C} \in \mathit{modifiesDom}(\delta) \end{cases}$$

where $\textsc{apply}(\delta(\mathtt{C}), \mathrm{CT}(\mathtt{C}))$, the application of the delta clause $\delta(\mathtt{C}) = \mathrm{DC} = \textbf{modifies}\ \mathtt{C} \cdots \{\cdots\}$ to the class definition $\mathrm{CT}(\mathrm{C}) = \mathrm{CD}$, is the class definition CD' defined as follows:

$$\mathrm{CD}'(\textbf{extends}) = \begin{cases} \mathrm{CD}(\textbf{extends}) & \text{if } \mathrm{DC}(\textbf{extending}) = \emptyset \\ \mathrm{C}' & \text{if } \mathrm{DC}(\textbf{extending}) = \{\mathrm{C}'\} \end{cases}$$

$$\mathrm{CD}'(\mathrm{a}) = \begin{cases} \mathrm{CD}(\mathrm{a}) & \text{if } \mathrm{a} \notin \mathit{dom}(\mathrm{DC}) \\ \mathrm{AD} & \text{if } \mathrm{DC}(\mathrm{a}) = \textbf{adds}\ \mathrm{AD} \\ \mathtt{MD}[\overline{\mathrm{s}}/\textbf{original}()] & \text{if } \mathrm{DC}(\mathrm{a}) = \textbf{modifies}\ \mathtt{MD} \\ & \quad \text{and } \mathrm{CD}(\mathrm{a}) = \cdots \mathrm{a}(\cdots)\{\overline{\mathrm{s}};\ \textbf{return}\ \mathrm{y};\} \end{cases}$$

The semantics of the **original**() call is captured by replacing the occurrence of **original**() in the method body specified by the modifies subclause with the body of the unmodified method.

For any given total order of delta module application, a LPΔJ SPL defines a *product generation mapping*. That is, a partial mapping from each feature configuration $\overline{\psi}$ in Φ to the class table of the product that is obtained by applying the delta modules $\Delta(\overline{\psi})$ to the empty class table according to the given order. The product generation mapping can be partial since a non-applicable delta module may be encountered during product generation such that the resulting product is undefined.

Unambiguous and Type-Safe* LPΔJ *Product Lines A LPΔJ SPL is *unambiguous* if all total orders of delta modules that are compatible with the application partial order define the same product generation mapping. In an unambiguous SPL, for every feature configuration at most one product implementation is generated.

In order to find a criterion for unambiguity, we define the notion of compatibility of a set of delta modules. A set of delta modules is called *compatible* if no class added or removed in one delta module is added, removed or modified in another delta module contained in the same set, and for every class modified in more than one delta module, its direct superclass is changed at most by one delta clause and the fields and methods added, modified or removed are distinct. For a set of compatible delta modules, any order of delta module application yields the same class table since the alterations in compatible delta modules do not interfere with each other.

A SPL is *locally unambiguous* if every set *S* of delta modules in the partition of *dom*(DMT) provided by the application partial order $<_{\mathrm{DMT}}$ is compatible. If the SPL L is locally unambiguous, then it is unambiguous. Local unambiguity can be checked by inspecting the delta modules in each partition only once.

A LPΔJ SPL is *type-safe* if the following conditions hold: (i) its product generation mapping is total, (ii) it is locally unambiguous, and (iii) all generated products are well-typed LJ programs.

4. Generalization of FOP

In this section, we show that Pure DOP is generalization of FOP [7] by providing a mapping from LFJ [13] into LPΔJ.

4.1 Recalling LFJ

The syntax of the LFJ extensions to LJ is given in Figure 3. It is taken from [13]. A feature module definition FMD contains the

$$
\begin{aligned}
&\llbracket \textbf{feature}\ \varphi\ \{\overline{\mathrm{cd}}\ \ \overline{\mathrm{rcd}}\}\rrbracket = \\
&\qquad \textbf{delta}\ \varphi\ \{\ \overline{\textbf{adds}\ \mathrm{cd}}\ \ \overline{\llbracket \mathrm{rcd}\rrbracket}\ \} \\
&\llbracket \textbf{refines class}\ \mathrm{C}\ \textbf{extending}\ \mathrm{C}\ \{\ \overline{\mathrm{fd}};\ \ \overline{\mathrm{md}}\ \ \overline{\mathrm{rmd}}\ \}\rrbracket = \\
&\qquad \textbf{modifies}\ \mathrm{C}\ \textbf{extending}\ \mathrm{C}\ \{\ \overline{\textbf{adds}\ \mathrm{fd}}\ \ \overline{\textbf{adds}\ \mathrm{md}}\ \ \overline{\llbracket \mathrm{rmd}\rrbracket}\ \} \\
&\llbracket \textbf{refines}\ \mathrm{ms}\ \{\bar{\mathrm{s}};\ \textbf{Super}();\ \bar{\mathrm{s}};\ \textbf{return}\ \mathrm{y};\}\rrbracket = \\
&\qquad \textbf{modifies}\ \mathrm{ms}\ \{\bar{\mathrm{s}};\ \textbf{original}();\ \bar{\mathrm{s}};\ \textbf{return}\ \mathrm{y};\}
\end{aligned}
$$

Figure 4. Translation of a feature module to a delta module

feature φ and a set of class definitions $\overline{\mathrm{cd}}$ and class refinement definitions $\overline{\mathrm{rcd}}$. Class definitions are given according to the syntax of LJ. A class refinement definition can change the superclass, add fields $\overline{\mathrm{fd}}$, provide new method definitions $\overline{\mathrm{md}}$ and refine existing method definitions $\overline{\mathrm{rmd}}$. A method refinement can wrap the existing method body using the **Super**() construct.

A *feature module table* FMT is a mapping from feature names to feature module definitions. A LFJ product line can be described by a 3-tuple $\mathrm{L} = (\mathrm{FMT}, \Phi, <_{\mathrm{FMT}})$ consisting of:

1. a feature module table FMT with a feature module for each feature of the SPL,
2. the set of the valid feature configurations $\Phi \subseteq \mathscr{P}(dom(\mathrm{FMT}))$,
3. a total order $<_{\mathrm{FMT}}$ on the set of features $dom(\mathrm{FMT})$.

The product associated to a feature configuration $\overline{\psi}$ is generated by composing (see Section 3.1 of [13]) the feature modules associated to the features in $\overline{\psi}$ according to the total order $<_{\mathrm{FMT}}$. During feature module composition, newly defined classes, fields and methods are added and class and method refinements are carried out. According to [13], a LFJ product line is *type-safe* if all generated products are well-typed LJ programs.

4.2 Mapping LFJ into LPΔJ

A product line in FOP can be represented as a product line in Pure DOP. The set of features and the set of valid feature configurations in both product lines is the same. Every feature module in a LFJ product line is mapped to a delta module where additions are translated to adds clauses and refinements to modifies clauses. The application condition of the delta module denotes all configurations in which the respective feature is contained. The ordering of delta module application is the total ordering of the feature modules.

Formally, the mapping from LFJ product lines to LPΔJ product lines is defined as follows: for a LFJ product line $\mathrm{L} = (\mathrm{FMT}, \Phi, <_{\mathrm{FMT}})$, $\llbracket \mathrm{L} \rrbracket$ denotes the corresponding LPΔJ product line $(\overline{\varphi}, \Phi, \mathrm{DMT}, \Gamma, <_{\mathrm{DMT}})$ where

- $\overline{\varphi} = dom(\mathrm{FMT}) = dom(\mathrm{DMT})$,
- The delta module table DMT is obtained by translating each feature module in FMT to a delta module with the same name, according to the clauses in Figure 4,
- $\Gamma : dom(\mathrm{DMT}) \to \Phi$, where $\Gamma(\varphi) = \{\overline{\psi} \mid \overline{\psi} \in \Phi \text{ and } \varphi \in \overline{\psi}\}$,
- $<_{\mathrm{DMT}}$ is the total order on $\{\{\varphi\} \mid \varphi \in \overline{\varphi}\}$ defined by: $\{\varphi_1\} <_{\mathrm{DMT}} \{\varphi_2\}$ if and only if $\varphi_1 <_{\mathrm{FMT}} \varphi_2$.

The following theorem states that the LPΔJ product lines generates the same products as the LFJ product line. Hence, Pure DOP is a true generalization of FOP.

THEOREM 4.1. *If* L *is a type safe* LFJ *product line, then* $\llbracket \mathrm{L} \rrbracket$ *is a type safe* LPΔJ *product line such that, for every valid feature configuration* $\overline{\psi}$*, the product for* $\overline{\psi}$ *generated by* L *is the same as the product for* $\overline{\psi}$ *generated by* $\llbracket \mathrm{L} \rrbracket$.

Although it is possible in principle to encode FOP in Core DOP, a straightforward embedding as for Pure DOP is not possible. This is because a feature-oriented SPL may have several base feature modules, while Core DOP requires exactly one core module as starting point for product generation.

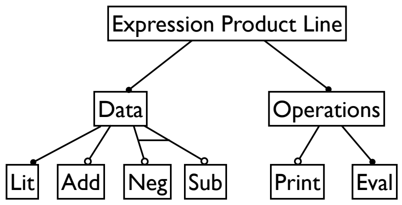


Figure 5. Feature model for evolved Expression Product Line

5. Pure DOP for Product Line Development

Pure DOP supports proactive, extractive and reactive product line development [22]. In the proactive approach, the scope of the product line, i.e., the set of products to be developed, is analyzed beforehand. All reusable artifacts are planned and developed in advance. The example for Pure DOP presented in Section 2 can be seen as proactive product line development, since we start from the feature model defining the scope of the product line and develop delta modules and a Pure DOP SPL for these products. However, proactive development requires a high upfront investment to define the scope of the product line and to develop reusable artifacts.

Hence, in order to reduce the adoption barrier for product line engineering, Krueger [22] proposes the usage of reactive and extractive approaches. In reactive product line engineering, only a basic set of products is developed. When new customer requirements arise, the existing product line is evolved. The extractive approach allows turning a set of existing legacy application into a product line. Development starts with the existing products from which the other products of the product line are derived.

FOP [7, 13] supports proactive product line development well. However, since feature modules are restricted to add or refine existing classes, FOP does not support extractive development and only partially supports reactive development. It is not possible to start from an existing legacy application comprising a larger set of features and to remove features. Moreover, in order to deal with new requirements following the reactive approach, feature modules might have to be refactored to remove functionalities. Also, in Core DOP, extractive product line development is not straight forward, since one product has to be chosen as designated core product. In contrast, Pure DOP is flexible and expressive enough to cover all three product line engineering approaches directly.

5.1 Reactive Product Line Engineering

In reactive product line engineering, development starts with an initial product line that is evolved in order to deal with changing customer requirements. Consider as initial product line the example depicted in Listing 5. Assume now that a new feature Sub should be introduced for representing subtraction expressions. In the new EPL, the Sub feature should be an alternative to the Neg feature. Additionally, the Print feature should become optional and the Eval feature mandatory. The feature diagram for the evolved product line is given in Figure 5.

In order to realize the new Sub feature, we have to add delta modules that introduce the corresponding data structure for subtraction and the associated print and the evaluation functionalities. The respective delta modules are shown in Listing 6. The specification for the evolved SPL is shown in Listing 7, where the operator $\mathbf{choose1}(P_1, \ldots, P_n)$ means at most one of the propositions $P_1, \ldots, P_n$ is true (see [5]).

```
delta DSub  {
  adds class Sub implements Exp {
    Exp expr1;
    Exp expr 2:
    Sub(Exp a, Exp b) { expr1 = a;  expr2= b; }
  }
}

delta DSubPrint {
  modifies Sub {
    adds void print() { expr1.print(); System.out.print("-");  expr2.print();}
  }
}

delta DSubEval{
   modifies class Sub {
     adds int eval() { return expr1.eval() − expr2.eval(); }
   }
}
```

Listing 6: Delta modules for Sub feature

```
features Lit, Add, Neg, Sub, Print, Eval
configurations Lit & Eval & choose1(Neg,Sub)
deltas
  [ DLit,
    DAdd when Add,
    DNeg when Neg,
    DSub when Sub ]

  [ DLitPrint when Print,
    DLitEval,
    DAddPrint when (Add & Print),
    DAddEval when Add,
    DNegPrint when (Neg & Print),
    DNegEval when Neg,
    DSubPrint when (Sub & Print),
    DSubEval when Sub ]
```

Listing 7: Pure DOP specification of the evolved EPL

As we can see in this example, Pure DOP supports reactive product line development, first, by adding new delta modules to implement new product features or to deal with new feature combinations, and, second, by reconfiguring the application conditions and the delta module order in the product line configuration to capture changes in the feature model.

5.2 Extractive Product Line Engineering

Extractive product line engineering starts with a set of existing legacy applications from which the other products of the product line are generated. Assume that we have already developed a product containing the Lit, Neg and Print features and a product containing the Lit, Add and Print features. Now, we want to transform these existing legacy applications into a product line according to the feature model in Figure 1.

First, the existing applications have to be transformed into delta modules that are applied initially. Listing 8 shows two delta modules adding the implementation of the two existing products, respectively. Second, in order to provide product implementations with less features, delta modules have to be specified that remove functionality from the existing products. Listing 9 shows the delta module that removes the feature Add.

Listing 10 shows the extractive implementation of the product line described by the feature model in Figure 1 starting from a product with features Lit, Neg and Print and a product with features Lit, Add, and Print introduced by the delta modules `DLitNegPrint` and `DLitAddPrint` in the first and second partitions, respectively. Their application conditions are exclusive such that for any feature configuration product generation starts with one of them. If the Add

```
delta DLitNegPrint{
  adds interface Exp {
    void print();
  }
  adds class Lit implements Exp {
    int value;
    Lit(int n) { value = n; }
    void print() { System.out.println(value); }
  }
  adds class Neg implements Exp {
   Exp expr;
   Neg(Exp a) { expr1 = a; }
   void print() { System.out.print("-("); expr.print(); System.out.print(")");}
  }
}

delta DLitAddPrint{
  adds interface Exp {
    void print();
  }
  adds class Lit implements Exp {
    int value;
    Lit(int n) { value = n; }
    void print() { System.out.println(value); }
  }
  adds class Add implements Exp {
    Exp expr1;
    Exp expr2;
    Add(Exp a, Exp b) { expr1 = a; expr2 = b; }
    void print() { expr1.print(); System.out.print(" + "); expr2.print(); }
  }
}
```

Listing 8: Delta modules introducing the two legacy products

```
delta DremAdd  {
  remove Add
}
```

Listing 9: Delta module removing the Add feature

```
features Lit, Add, Neg, Print, Eval
configurations Lit & Print
deltas
  [ DLitNegPrint when (!Add & Neg) ]

  [ DLitAddPrint when (Add | !Neg) ]

  [ DNeg when (Add & Neg),
    DremAdd when (!Add & !Neg) ]

  [ DNegPrint when (Add & Neg),
    DLitEval when Eval,
    DAddEval when (Add & Eval),
    DNegEval when (Neg & Eval) ]
```

Listing 10: Pure DOP specification of the extractive EPL

feature is not selected and the Neg feature is selected, we start with the existing product in delta module `DLitNegPrint`. Otherwise, we start with the existing product in delta module `DLitAddPrint`. If both features Add and Neg are selected, we add the Neg feature by the delta modules `DNeg` and `DNegPrint` of Listing 4. If both the Add feature and the Neg feature are not selected, we remove the Add feature by the delta module `DremAdd` of Listing 9. Finally, we add the evaluation functionality if the feature Eval is selected.

This example shows that Pure DOP supports extractive product line engineering by introducing the existing products in initial delta modules, by delta modules removing functionality, and by specifying the product line to generate the products from the existing products by suitable delta module application.

```
delta DremPrintLit {
  modifies interface Exp { removes print }
  modifies class Lit { removes print }
}

delta DremPrintAdd {
  modifies class Add { removes print }
}

delta DremPrintNeg {
  modifies class Neg { removes print }
}
```

Listing 11: Delta modules removing the Print feature

```
features Lit, Add, Neg, Sub, Print, Eval
configurations Lit & Eval & choose1(Neg,Sub)
deltas
  [ DLitNegPrint when (!Add & Neg),
    DSub when Sub ]

  [ DLitAddPrint when (Add | !Neg) ]

  [ DNeg when (Add & Neg),
    DremAdd when (!Add & !Neg)]

  [ DNegPrint when (Add & Neg & Print),
    DLitEval,
    DAddEval when Add,
    DNegEval when Neg,
    DremPrintLit when !Print,
    DremPrintAdd when (!Print & Add),
    DremPrintNeg when (!Print & Neg),
    DSubPrint when (Sub & Print),
    DSubEval when Sub ]
```

Listing 12: Pure DOP specification of the evolved extractive EPL

5.3 Combining Extractive and Reactive PL Engineering

Extractive and reactive product line engineering can be combined. An initial product line is developed from a set of existing legacy applications and evolved when new requirements arise. Consider, the product line developed using the extractive approach in Listing 10. Assume, that now the Sub feature should be added and the product line should be changed to implement the feature diagram in Figure 5. Since in this product line, the feature Print is optional, we have to provide delta modules that remove the printing functionality from the `Lit`, `Add` and `Neg` classes. These delta modules are depicted in Listing 11.

Listing 12 shows the specification of the evolved product line depicted in Listing 10. The generation starts again from the two delta modules `DLitNegPrint` and `DLitAddPrint` introducing the existing products. Additionally, the product line contains delta modules for adding the Sub feature (cf. Listing 6) and delta modules for removing the Print feature (cf. Listing 11).

6. Comparison with Core DOP

In the traditional presentation of DOP [29], which we refer to as Core DOP, program generation always starts from a core module containing the implementation of a selected valid product of the product line. Then, delta modules specify the changes to the core module in order to implement the other products. Moreover, in the presentation of Core DOP given in [29]:

- the feature configuration corresponding to the product implemented by the core module is specified in the code of the core module,
- the application condition of a delta module is specified in the code of the delta module by a clause of the form "**when** γ", where γ is a propositional constraint specifying the feature configurations in which the delta module has to be applied, and
- the application partial order for the delta modules is specified in the code of the delta modules using a clause of the form "**after** $\overline{\delta}$", which specifies that the delta module must be applied after all applicable delta modules in $\overline{\delta}$ have been applied.

Pure DOP and Core DOP are indeed equivalent:

- A Pure DOP product line can be expressed as a Core DOP product line by adding an empty product to the product line and choosing it as the product implemented by the core module.
- A Core DOP product line can be expressed as a Pure DOP product line by transforming the core module into a delta module that has to be applied before any other delta module for all the valid feature configurations.

Pure DOP is a conceptual simplification of Core DOP dropping the notion of the core module and separating the specification of the application conditions and of the application ordering from the delta modules. This presents the following advantages:

- Pure DOP allows reusing delta modules for implementing different product lines (cf. Sections 2 and 5).
- Every delta module in Pure DOP containing only adds clauses can play the role of the core module. Thus, product lines with multiple base modules, that may not correspond to valid products, are possible. As a consequence, Pure DOP is a true generalization of FOP (cf. Section 4).
- Pure DOP supports the evolution of product lines. If a product line evolves such that the core product of a Core DOP product line is no longer a valid product, the core module and potentially all delta modules have to be refactored. In contrast, in pure DOP, existing delta modules can be reused for the specification of the evolved product line (cf. Section 5).

7. Related Work

The notion of program deltas is introduced in [23] to describe the modifications of object-oriented programs. In [30], delta-oriented modeling is used to develop product line artifacts suitable for automated product derivation and implemented with frame technology [36]. This approach is extended in [28] to a seamless delta-oriented model-based development approach for SPLs. In [11], an algebraic representation of delta-oriented product lines is presented. The main focus in [11] is to reason about conflicting modifications and to devise a general criterion to guarantee the unambiguity of product lines using conflict-resolving deltas. The unambiguity property presented in this work is an instance of the criterion presented in [11], but it is more restrictive since it requires to order all potential conflicts. Delta modules are one possibility to implement arrows in the category-theoretical framework for program generation proposed by Batory in [6].

Feature-oriented programming (FOP) [2, 7, 13, 34], Core DOP [29] and Pure DOP are compositional approaches [19] for implementing SPLs. For a detailed comparison between FOP and Core DOP, the reader is referred to [29]. Other compositional approaches used to implement product lines rely on aspects [18], framed aspects [24], combinations of feature modules and aspects [3, 25], mixins [31], hyperslices [33] or traits [8, 14]. In [23], several of these modern program modularization techniques are compared with respect to their ability to represent feature-based variability. Furthermore, the modularity concepts of recent languages, such as SCALA [26] or NEWSPEAK [10], can be used to represent product features.

In [1], an approach is presented that combines reactive and extractive product line engineering [22] based on aspect-oriented program refactorings. The modification operations that can be specified in delta modules are sufficient to express before, after and around advice considered in aspect-oriented programming [21]. Delta modules do not comprise a specification formalism for modifications to be carried out at several places of a program (such as pointcuts), such that all program modifications have to be explicitly specified. Adding a pointcut-specification technique to delta modules would allow encoding AOP by DOP, which is a subject of future work. However, delta modules are more flexible than aspects by their ability to remove functionality, such that a program refactoring is not required to evolve a product line when functionality has to be removed.

8. Conclusions and Future Work

In this paper, we have proposed *pure delta-oriented programming* (Pure DOP) as a conceptual simplification of Core DOP [29]. An implementation of the Pure DOP programming language presented in this paper and a core calculus with a constraint-based type system are currently being developed. Following the conceptual comparison of FOP, Core DOP and Pure DOP in this paper, we are evaluating Pure DOP empirically at larger case examples and investigating the extraction of delta modules from version histories.

The concept of Pure DOP is not bound to a particular programming language. In this work, we have instantiated it for LFJ. For future work, we are aiming to use other languages for the underlying product implementations. A starting point is the trait-based calculus FEATHERWEIGHT RECORD-TRAIT JAVA (FRTJ) [8, 9]. In FRTJ, classes are assembled from interfaces, records (providing fields) and traits [14] (providing methods) that can be directly manipulated by designated composition operations. These operations make FRTJ a good candidate for implementing delta modules in an expressive way.

References

[1] V. Alves, P. Matos, L. Cole, A. Vasconcelos, P. Borba, and G. Ramalho. Extracting and evolving code in product lines with aspect-oriented programming. In *Transactions on aspect-oriented software development IV*, pages 117–142. Springer-Verlag, 2007.

[2] S. Apel, C. Kästner, and C. Lengauer. Feature Featherweight Java: A Calculus for Feature-Oriented Programming and Stepwise Refinement. In *GPCE*, pages 101–112. ACM, 2008.

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

[4] P. G. Bassett. *Framing software reuse: lessons from the real world*. Prentice-Hall, Inc., 1997.

[5] D. Batory. Feature Models, Grammars, and Propositional Formulas. In *SPLC*, volume 3714 of *LNCS*, pages 7–20. Springer, 2005.

[6] D. Batory. Using modern mathematics as an FOSD modeling language. In *GPCE*, pages 35–44. ACM, 2008.

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

[8] L. Bettini, F. Damiani, and I. Schaefer. Implementing Software Product Lines using Traits. In *SAC, OOPS Track*, pages 2096–2102. ACM, 2010.

[9] L. Bettini, F. Damiani, I. Schaefer, and F. Strocco. A Prototypical Java-like Language with Records and Traits. In *PPPJ*. ACM, 2010.

[10] G. Bracha. Executable Grammars in Newspeak. *ENTCS*, 193:3–18, 2007.

[11] D. Clarke, M. Helvensteijn, and I. Schaefer. Abstract delta modeling. In *Proc. of GPCE*, 2010. (to appear).

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

[13] B. Delaware, W. Cook, and D. Batory. A Machine-Checked Model of Safe Composition. In *FOAL*, pages 31–35. ACM, 2009.

[14] S. Ducasse, O. Nierstrasz, N. Schärli, R. Wuyts, and A. Black. Traits: A mechanism for fine-grained reuse. *ACM TOPLAS*, 28(2):331–388, 2006.

[15] A. Igarashi, B. Pierce, and P. Wadler. Featherweight Java: A minimal core calculus for Java and GJ. *ACM TOPLAS*, 23(3):396–450, 2001.

[16] K. C. Kang, S. G. Cohen, J. A. Hess, W. E. Novak, and A. S. Peterson. Feature-Oriented Domain Analysis (FODA) Feasibility Study. Technical report, Carnegie Mellon Software Engineering Institute, 1990.

[17] C. Kästner and S. Apel. Type-Checking Software Product Lines - A Formal Approach. In *ASE*, pages 258–267. IEEE, 2008.

[18] C. Kästner, S. Apel, and D. Batory. A Case Study Implementing Features Using AspectJ. In *SPLC*, pages 223–232. IEEE, 2007.

[19] C. Kästner, S. Apel, and M. Kuhlemann. Granularity in Software Product Lines. In *ICSE*, pages 311–320. ACM, 2008.

[20] 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 *SPLC*. IEEE, 2009.

[21] G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, and W. G. Griswold. An Overview of AspectJ. In *ECOOP*, volume 2072 of *LNCS*, pages 327–353. Springer, 2001.

[22] C. Krueger. Eliminating the Adoption Barrier. *IEEE Software*, 19(4):29–31, 2002.

[23] R. Lopez-Herrejon, D. Batory, and W. Cook. Evaluating Support for Features in Advanced Modularization Technologies. In *ECOOP*, volume 3586 of *LNCS*, pages 169–194. Springer, 2005.

[24] N. Loughran and A. Rashid. Framed aspects: Supporting variability and configurability for aop. In *ICSR*, volume 3107 of *LNCS*, pages 127–140. Springer, 2004.

[25] M. Mezini and K. Ostermann. Variability management with feature-oriented programming and aspects. In *SIGSOFT FSE*, pages 127–136. ACM, 2004.

[26] M. Odersky. The Scala Language Specification, version 2.4. Technical report, Programming Methods Laboratory, EPFL, 2007.

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

[28] I. Schaefer. Variability Modelling for Model-Driven Development of Software Product Lines. In *Intl. Workshop on Variability Modelling of Software-intensive Systems*, 2010.

[29] I. Schaefer, L. Bettini, V. Bono, F. Damiani, and N. Tanzarella. Delta-oriented Programming of Software Product Lines. In *SPLC*, volume 6287 of *LNCS*. Springer, 2010.

[30] I. Schaefer, A. Worret, and A. Poetzsch-Heffter. A Model-Based Framework for Automated Product Derivation. In *Proc. of MAPLE*, 2009.

[31] Y. Smaragdakis and D. Batory. Mixin layers: an object-oriented implementation technique for refinements and collaboration-based designs. *ACM Trans. Softw. Eng. Methodol.*, 11(2):215–255, 2002.

[32] R. Strniša, P. Sewell, and M. Parkinson. The Java module system: core design and semantic definition. In *OOPSLA*, pages 499–514. ACM, 2007.

[33] P. Tarr, H. Ossher, W. Harrison, and S. M. Sutton, Jr. N degrees of separation: multi-dimensional separation of concerns. In *ICSE*, pages 107–119, 1999.

[34] S. Thaker, D. Batory, D. Kitchin, and W. Cook. Safe Composition of Product Lines. In *GPCE*, pages 95–104. ACM, 2007.

[35] M. Torgersen. The Expression Problem Revisited. In *ECOOP*, volume 3086 of *LNCS*, pages 123–146. Springer, 2004.

[36] H. Zhang and S. Jarzabek. An XVCL-based Approach to Software Product Line Development. In *Software Engineering and Knowledge Engineering*, pages 267–275, 2003.

Improving Reuse of Component Families by Generating Component Hierarchies

Marko Rosenmüller
School of Computer Science,
University of Magdeburg,
Germany
rosenmue@ovgu.de

Norbert Siegmund
School of Computer Science,
University of Magdeburg,
Germany
nsiegmun@ovgu.de

Martin Kuhlemann
School of Computer Science,
University of Magdeburg,
Germany
mkuhlema@ovgu.de

ABSTRACT

Feature-oriented software development (FOSD) enables developers to generate families of similar components. However, current FOSD approaches degrade component reuse because they do not allow a developer to combine multiple components of the same family in a larger program. This is because individual family members cannot be distinguished from each other. We present an approach to model and generate *component hierarchies* that allow a programmer to combine multiple component variants. A component hierarchy structures the components of a family according to their functionality. Due to subtyping between the components of a hierarchy, client developers can write generic code that works with different component variants.

Categories and Subject Descriptors

D.2.3 [**Software Engineering**]: Coding Tools and Techniques; D.2.13 [**Software Engineering**]: Reusable Software—*Domain engineering, Reusable libraries*

General Terms

Design, Languages

1. INTRODUCTION

Scalable software reuse can be achieved by developing components, libraries, and frameworks (in the following referred to as components) as *software product lines (SPLs)* [4]. From a component SPL, programmers can derive a family of similar components that can be distinguished in terms of *features* [6]. Features represent characteristics of a component that are of interest to some stakeholder. A developer can build more complex SPLs by combining multiple component SPLs. This results in a set of interdependent SPLs which we call a *multi product line (MPL)* [21].

In previous work, we presented an approach to model MPLs and to automate their configuration [21]. In this paper, we extend the modeling approach and address the implementation of MPLs with *feature-oriented programming (FOP)* [19, 3]. FOP and other approaches for SPL development do not allow a programmer to use different variants of a component in the same program. The reason is that we cannot distinguish two variants of a component from each other since they are derived from the same code base and use the same names (e.g., for classes). For example, in the customizable DBMS Berkeley DB[1], programmers use the C preprocessor to implement variability. Since a function has the same name in all Berkeley DB variants, we cannot use different variants of the DBMS in the same program (e.g., one for stream processing and one for persistent storage).

As another example consider an SPL for container data structures. We can implement it with FOP to derive different kinds of data structures (e.g., a sorted list and a synchronized list). A client developer should be able to use different variants of the SPL in the same program. Ideally, the data structures should span a type hierarchy to simplify development of generic client code. For example, a sorted list and a synchronized list should be subtypes of a basic list to use them polymorphically. OOP concepts, such as inheritance (e.g., creating a subclass for each kind of list), cannot solve the problem because a programmer has to create subclasses for every class of each component variant [4]. Finally, most FOP approaches completely replicate the code when generating different variants of a component. We summarize the observed problems as follows:

Naming Conflicts. When deriving different variants of a component from an SPL, the names of classes in different variants are the same. Hence, a programmer (and a compiler) cannot distinguish the different variants (e.g., to create objects).

Missing Subtyping. Even though different SPL instances provide a similar interface there is no subtyping relationship between them. This hampers development of generic code in client applications.

Code Replication. Different variants of the same SPL share functionality but there is no code reuse: features that are used in two variants are completely replicated.

We observe these problems when components are embedded in a program (e.g., as statically linked libraries) and used via an API. Hence, we focus on components that are represented in the programming language (e.g., as a set of classes). We define three requirements to enable reuse of different variants of a component within the same program:

[1]http://www.oracle.com/database/berkeley-db/

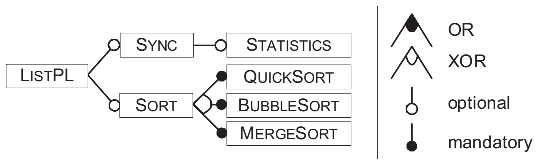


Figure 1: Feature diagram of an SPL for list data structures.

1. Instance identification: programmers must be able to distinguish two instances of the same SPL.
2. Subtyping: SPL instances should span a type hierarchy to be used polymorphically.
3. Code Reuse: code should be reused between instances of the same SPL.

Requirement (1) is mandatory for using different component variants in the same program. Requirements (2) and (3) are optional: Requirement (2) simplifies client development because it allows a programmer to write generic code; Requirement (3) is an optimization.

To address the problems already at a conceptual, implementation-independent level, we extend our approach for modeling MPLs. We use *SPL specialization* [7] to define *component hierarchies* and define a subtype relationship between component variants. We then demonstrate how component hierarchies can be generated from FOP code.

2. COMPONENT SPLS

A component SPL allows a programmer to derive a component tailored to her needs. As a running example, consider an SPL for list data structures (ListPL). We can use it to derive different kinds of linked lists such as sorted and synchronized lists. In Figure 1, we show the ListPL *feature diagram* [11, 6], a hierarchical representation of the features. A feature diagram includes relations between features (such as an XOR relation between alternative features) to avoid invalid feature combinations in a concrete SPL instance. For example, feature Sort in Figure 1 represents the functionality for sorting list elements. It has three alternative subfeatures that implement different sort algorithms.

A programmer can use the ListPL to implement a mail client SPL (MailPL). The mail client may also use other SPLs such as a DBMS for mail storage. We call the whole set of interdependent SPLs a *multi product line (MPL)* [21]. The product of an MPL is a set of interacting products of the underlying SPLs. With an *MPL composition model* we describe which SPL instances are used within an MPL. An example for the mail client is shown in Figure 2: MailPL uses an instance of DbmsPL and two different instances of ListPL, which we describe with a composition relationship. SPL instance names (e.g., `sortList`) are used to identify different instances of the same SPL on the model level.

2.1 SPL Specialization Hierarchies

In order to specify which functionality an SPL instance (e.g., `sortList` in Fig. 2) must provide, we use SPL specializations. A specialization of an SPL is a configuration step that eliminates configuration choices [7]. Usually, a specialization does not specify an SPL instance completely; it is only a partial configuration that still provides some variability. In Figure 3, we show an extended ListPL model. We use inheritance to denote specializations of `ListPL`, which results in an *specialization hierarchy*. For example, `SortListPL` and `SyncListPL` are specializations of `ListPL`, each representing a subset of the variants. Feature Sort is included in all instances of `SortListPL`. Hence, we can only derive sorted lists from it. `QuickSortListPL` and `MergeSortListPL` are specializations of `SortListPL` that implement different sorting algorithms. A specialization step does not necessarily add features to an SPL. For example, it may explicitly exclude a feature. In general, arbitrary constraints can be used to create a specialized SPL by reducing the number of valid configurations.

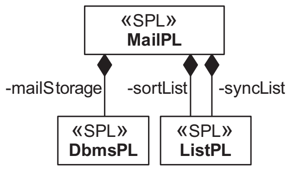


Figure 2: Modeling product lines with UML: A mail client SPL (MailPL) using a DBMS and List SPL.

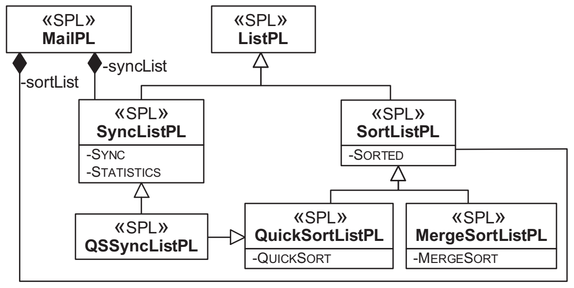


Figure 3: Specialization hierarchy of SPL ListPL. MailPL uses two specializations of ListPL.

A *fully specialized* SPL represents only a single configuration [7]. We can use it to directly derive the corresponding SPL instance. In contrast, when creating an instance from an incompletely specialized SPL, we have to bind remaining variability first. For example, we can create an instance from `SortListPL` by selecting feature QuickSort and excluding feature Sync.

2.2 Subtyping and SPL Interfaces

The specialization hierarchy defines a subtype relationship between SPLs: A specialized SPL D is a subtype of a less specialized SPL B. If an SPL is a specialization of another SPL, and thus a subtype, can be checked with a SAT solver [23]. Subtyping between SPLs allows us to use them polymorphically. For example, MailPL in Figure 3 uses `SortListPL`, but also accepts every subtype thereof such as an instance of `QuickSortListPL`. Hence, an SPL instance can be used at places where its super type is required.

To describe the interaction between SPLs, we introduce the notion of an *SPL interface*. We distinguish an SPL's *semantic interface* from its *programming interface* (which we introduce in Section 3.3). We define the semantic interface of a (specialized) SPL as the set of features that are present in all valid instances of the SPL (i.e., the minimally required features). These are mandatory features, features selected via specialization, and features required due to constraints. By adding features in specialization steps we extend the interface of an SPL. However, not every specialization step extends the interface. For example, the interface does usually not change when we add a constraint that excludes a feature. Hence, when the semantic interface of an SPL D (i.e., the set of features) is a superset of the the interface of an SPL B then D is a subytpe of B. On the other hand, when D is a subtype of B then the interface of D is a superset of the interface of B, but not necessarily a *proper* superset.

The expected and required semantic interfaces (i.e., the set of expected and required features) can be used to check whether one component provides the functionality required by another component. For example, we can check whether an instance of ListPL provides all features an instance of MailPL expects. This is a kind of semantic compatibility which is in contrast to the syntactic compatibility that is checked with programming interfaces.

2.3 Summary

Composition models and specialization hierarchies provide means to model MPLs and to distinguish different variants of an SPL at a conceptual level. The requirement to distinguish different instances of an SPL (Req. 1 in Sec. 1) is satisfied by using named SPL instances. The subtype relationship between SPL specializations allows us to use different instances polymorhically (Req. 2). Requirement 3 (Code Reuse) does not apply to the model level. Nevertheless, we can reuse SPL instances (i.e., an SPL's configuration) at different places in an MPL model and thus avoid redefinitions. In the next Section, we show how these concepts can be mapped to the implementation of an SPL.

3. GENERATING COMPONENTS

We demonstrate how the concepts can be applied to SPL implementation for the programming languages *Jak*[2] and *FeatureC++*[3], which are FOP extensions for Java and C++.

3.1 Feature-oriented Programming

FOP allows a programmer to implement the features of an SPL as separate *feature modules*. [19, 3]. Feature modules decompose a class into a base class and *class refinements*. In Figure 4, we depict the FeatureC++ code of the base implementation of a class `List` (Lines 1–6) of the ListPL, and two refinements of the class (Lines 7–16). Elements are added to the list via method `add`. The refinements in features SORT and SYNC extend the base implementation. They override method `add` to implement sorted insertion (Line 8) and synchronization (Line 12). Feature SYNC also adds a new field `sync` to synchronize access to the list. Overridden methods are called with `super` as shown for feature SYNC (Line 14).

A user defines an SPL instance by selecting a set of features that satisfy her requirements. A generator composes the corresponding feature modules to yield a concrete list instance. Using the feature modules of Figure 4, we can generate a simple list using the base implementation only, but we can also use different combinations of the features (e.g., to derive a sorted synchronized list). Jak and FeatureC++

Feature BASE

```
1  //Basic implementation of class List
2  template <class T>
3  class List {
4    Node<T>* head;
5    void add(T elem) { ... /* append at end */ }
6  };
```

Feature SORT

```
7  refines class List {
8    void add(T elem) { ... /* sorted insert */ }
9  };
```

Feature SYNC

```
10 refines class List {
11   SyncObject sync;
12   void add(T elem) {
13     LockObject lock(sync);  //syncronize access
14     super::Add(m);          //add element
15   }
16 };
```

Figure 4: FeatureC++ code of class `List` decomposed along the features Sort and Sync.

support static composition of classes. This means that according to the feature selection, the code of all refinements is composed into a single class at compile time.

3.2 Component Hierarchies

We map the modeling concepts (i.e., named SPL instances, SPL specialization, and subtyping) to the implementation of SPLs by generating component hierarchies. Before we present implementation techniques, we review the requirements defined in Section 1 with respect to FOP.

Instance Identification. Indentifying different instances of the same SPL means to distinguish different variants of an *implementation class*[4] of these instances. For example, generating a sorted and a synchronized instance of ListPL results in different variants of class `List` (e.g., a sorted list in one instance and a synchronized list in the other instance; cf. Fig. 4). When using static composition mechanisms such as Jak, FeatureC++, or the C/C++ preprocessor, all variants of class `List` have the same name. This makes it impossible to identify the different variants of a class (e.g., for creating objects). Hence, the component generation process must create unique names for implementation classes of different SPL instances.

Subtyping. Mapping the specialization hierarchy to the implementation of an SPL means that generated components (i.e., the SPL instances) have to follow this hierarchy too. Hence, when an SPL `Base` is a super type of an SPL `Derived` then the whole set of implementation classes in `Base` should be a super type of the corresponding classes in `Derived`. This is also known as *Family Polymorphism* [8]. The resulting subtype relationship is needed to simplify client development and must be available in the client language. For example, FeatureC++ generates plain C++ code and we should be able to use a generated component in C++ clients. This requires a subtype relationship between implementation classes of different component variants in the generated C++ code. Since we can have different SPL spe-

[2]http://userweb.cs.utexas.edu/users/schwartz/
[3]http://fosd.de/fcc/

[4]We refer to the classes that are used to implement an SPL as *implementation classes.*

cializations for different application scenarios, the specialization hierarchy and thus the subtype relationship may be different for different client programs.

Code Reuse. Generating different variants of an SPL usually means code replication because the code of shared features is repeated for each instance. As a result, we get a similar increase in binary size as observed for C++ templates (a.k.a. *code bloat*). Since whole features are replicated between generated variants, the classes, methods, and refinements of a feature should be automatically reused across a family of SPL instances [14].

3.3 SPL Programming Interfaces

In Section 2.2, we defined the *semantic interface* of a (specialized) SPL. Based on this definition we define an SPL's *programming interface* as the union of the programming interfaces of the implementation classes defined in the features of the semantic interface. For example, the programming interface of `SortListPL` (cf. Fig. 3) consists of the interfaces of all classes defined in the base implementation of ListPL and feature Sort (cf. Fig. 1). It does not include classes or methods introduced by features QuickSort, MergeSort, or BubbleSort because only one of the features will be present in an instance. In specialization steps, we extend an SPL's programming interface up to a complete interface for a concrete component.

The subtype relationship between specialized SPLs also applies to the implementation classes. Hence, when SPL `Derived` is a specialization (and a subtype) of SPL `Base` then an implementation class `C` defined in `Derived` is also a subtype of class `C` defined in SPL `Base`. This means that a feature can only add members to the interface of a class but cannot modify members because it would conflict with the subtype relationship. For example, a feature cannot extend the signature of a method as it is sometimes done in preprocessor-based implementations of an SPL [12, 20]. We argue that such extensions must be avoided because they complicate SPL development and hamper use of SPL instances [20].

4. IMPLEMENTATION MECHANISMS

We present different mechanisms for generating component families that enable a programmer to use multiple instances of an SPL at the same time. We analyze each mechanism with respect to the presented requirements.

4.1 Namespaces / Packages

A simple way to distinguish sets of classes that have the same name is to group them into namespaces (C++) or packages (Java). The FeatureC++ compiler supports the generation of a package for each SPL instance. For example, we can use a namespace `SortList` to group all classes of a sorted list SPL instance. For Jak, this is possible with *refactoring feature modules* which move generated classes into a package [13]. Refactoring feature modules are a general mechanism that can be applied to other languages as well. In the following, we analyze the approach with respect to our requirements.

Instance Identification. We distinguish classes of different component variants via their package name. The package thus provides a unique type for each class. For example, we can define a client method that creates sorted lists:

```
class MailClient {
   sortList.List createList() {
      return new sortList.List();
   }
}
```

The name of package `sortList` corresponds to the name of the SPL instance defined in the MPL model (cf. Fig. 3). The instance name can thus be used for the code generation process. For example, the FeatureC++ compiler can generate required instances with their namespaces as defined in the composition model.

Subtyping. The namespace solution does not support subtyping between different variants of a class. The reason is that every generated SPL instance uses its own namespace and classes of one namespace are independent of the classes of a different namespace. For example, a class `quickSortList.List` is not a subtype of `sortList.List` even though both provide a similar interface. Furthermore, there is no representation of specialized SPLs in the namespace approach.

Code Reuse. There is no code reuse between classes of two SPL variants. For example, `quickSortList.List` and `mergeSortList.List` completely replicate the code of feature Sort and the base implementation. However, classes of different namespaces might be extracted and merged into a common class library [14]. This cannot avoid code replication completely but may be sufficient for many application scenarios. In contrast to the approach described in [14], static fields have to be handled differently to avoid shared state between different variants.

4.2 Virtual Classes

A namespace approach does not allow us to use implementation classes polymorhically. We can provide the required subtype relationship with *virtual classes* [15] as supported by CaesarJ[5] [1]. A virtual class is a nested class whose type depends on the type of an object of its enclosing class. In our case, the enclosing class represents a specialized SPL. With *mixin-based inheritance* [5], an enclosing CaesarJ class composes multiple classes. Mixin composition is similar to multiple inheritance but avoids some of its problems by linearizing the base classes. When implementing SPL features as enclosing classes, mixin composition can be used to compose features. The composition results in a specialized SPL that includes the features of all base classes. For example, in CaesarJ we define SortListPL (cf. Fig. 3) as follows[6]:

```
cclass SortListPL extends Sort & ListPL { }
```

`SortListPL` represents a specialized SPL that is defined via mixin composition of feature `Sort` with SPL `ListPL`.

Instance Identification. With virtual classes, a specialized SPL as well as an SPL instance is represented by a class. To use an SPL instance, we create an object of an SPL class (e.g., an instance of class `SortListPL`). The type of an implementation class, which is an inner virtual class, is defined by an object of an SPL class. For example, we can use an object of `SortListPL` to create sorted lists:

[5]http://caesarj.org

[6]CaesarJ classes are defined with keyword `cclass`.

```
cclass MailClient {
   SortListPL sortList = new QuickSortListPL();
   SortListPL.List createList() {
      return sortList.new List();
   }
}
```

In this example, `sortList` is an object of SPL instance `QuickSortListPL`. This specialized SPL has to correspond to a valid configuration. For example, we cannot create an instance of SortListPL because it does not provide a sorting implementation (cf. Fig. 3). The SPL instance object provides the `new` operator for creating objects of that instance. This is similar to the namespace approach. As in the namespace approach, we use the instance name `sortList` as defined in the MPL model.

Subtyping. The implementation classes of an SPL can be used polymorphically. For example, method

```
void display(SortListPL.List l)
```

accepts all kinds of sorted lists, which are defined in an SPL instance that is a subtype of `SortListPL`. Furthermore, due to the virtual class mechanism, a type can also be specified via an object. Hence, methods such as

```
void display(SortListPL plInst, plInst.List l)
```

can be used to ensure that an object (`plInst.List l`) corresponds to a particular SPL instance (`plInst`). In this example, the actual type of `plInst` could be `QuickSortListPL` (i.e., a subtype of `SortListPL`). List `l` then has to be an object of `QuickSortListPL.List`. This is used to distinguish objects of classes (e.g., `List`) of different SPL instances. With static type checking we can ensure that an object of one SPL instance is not passed to a different instance [9].

Code Reuse. The actual code reuse in a family of components depends on the concrete implementation of virtual classes. In CaesarJ, all implementation classes of a set of generated components form an inheritance hierarchy [1]. The hierarchy of a class corresponds to the refinement chain (i.e., the mixin list in CaesarJ) and is independent of the specialization hierarchy. This reduces code replication but does not completely avoid it: In a complex inheritance hierarchy, we have to replicate the code of refinements that are used multiple times at different positions in the hierarchy. However, this could be avoided with a different implementation.

Mixin Composition and Complexity Issues. The presented approach causes problems with respect to composition and complexity. The first issue is related to mixin composition. When creating a specialized SPL via mixin-based inheritance, we have to inherit from the SPLs as defined in our specialization hierarchy to achieve subtyping. At the same time, mixin composition is used to define the feature composition order: Features of the base classes are merged in the same order as they are listed in the base class definition. This entangles the subtype relationship and the feature composition order. Since feature composition is not commutative, it is impossible to achieve a valid feature order for all component hierarchies. A workaround in CaesarJ is to explicitly define the feature order as well as the parent SPLs required for subtyping. For example, we define a class `SyncSortListPL` (i.e., a synchronized sorted list), which should be a subtype of `SyncListPL` (cf. Fig 3) and `SortListPL` as:

```
cclass SyncSortListPL
   extends SyncListPL & SortListPL
         & Sync & Sort & ListBase { }
```

Here, `Sync`, `Sort`, and `ListBase` define the correct feature order; `SyncListPL` and `SortListPL` are used to define the required subtyping. However, mixin composition still increases the complexity of the SPL configuration process, which hinders its use for SPL development:

- Repeating the feature order for every SPL instance means additional configuration effort and is error-prone.
- An SPL instance is created by a user of an SPL (e.g., a developer of a client application) that does not know SPL implementation details such as the feature composition order.
- Changing the configuration of an inner component of the hierarchy (e.g., adding a feature to `SortListPL`; cf. Fig 3) is not possible without modifying every instance to explicitly define the feature order.
- The approach imposes an additional complexity on client developers due to the use of virtual classes. Hence, a client developer that uses only a single SPL instance is faced with an unneeded complexity.

Some of these issues can be solved by extending CaesarJ, e.g., by separating composition order from subclassing. We propose to address these complexity issues with a generative approach: based on an SPL implementation in an FOP language such as Jak, we generate virtual classes (e.g., CaesarJ code) including the specialization hierarchy with mixin-based inheritance. This generative approach avoids manual configuration via mixin composition. Furthermore, when only a single SPL instance is required, we generate plain Java code and avoid the complexity of virtual classes.

4.3 Generating SPL Interfaces

Both, the namespace approach and the virtual class approach, have drawbacks that limit their applicability. For a more general solution, we generate a hierarchy of SPL interfaces (i.e., the SPL's programming interface) to represent specialized SPLs. We use nested interfaces to represent the interfaces of SPL implementation classes. In Figure 5, we show an example for the generated interfaces of a subset of the ListPL hierarchy (Lines 1–15). A concrete SPL instance is defined as a class that implements the interface of the corresponding specialized SPL (Lines 16–20). This code transformation is similar to the implementation used in CaesarJ [1].

In contrast to the namespace approach, implementation classes of an instance are defined as nested classes within their instance (Lines 18–19). In contrast to virtual classes, the SPL specialization hierarchy is represented as a hierarchy of interfaces in the client language. We thus separate an SPL instance (i.e., a concrete implementation) from its interface. This corresponds to the fact that we can have an SPL specialization that does not correspond to a concrete instance. The interface can be used to define which functionality an SPL provides without a concrete implementation.

The generated interface hierarchy provides an emulation of virtual classes for plain Java and C++. As implemen-

SPL ListPL

```
 1 interface ListPL {
 2  abstract List newList();
 3  interface List {...}
 4  interface Node {...}
 5 }
```

SPL specialization SortListPL

```
 6 interface SortListPL extends ListPL {
 7  abstract List newList();
 8  interface List extends ListPL.List {...}
 9  interface Node extends ListPL.Node{...}
10 }
```

SPL specialization QuickSortListPL

```
11 interface QuickSortListPL extends SortListPL {
12  abstract List newList();
13  interface List extends ListPL.List {...}
14  interface Node extends ListPL.Node{...}
15 }
```

SPL instance QuickSortList

```
16 class QuickSortList implements QuickSortListPL {
17  List newList() {...}
18  class List implements QuickSortListPL.List {...}
19  class Node implements QuickSortListPL.Node{...}
20 }
```

Figure 5: Generated interface hierarchy representing SPL specialization (Lines 1–15) and a generated class representing an SPL instance (Lines 16–20).

tation mechanism, it is possible to use refactoring feature modules to extract the required interface of a specialized SPL. In the following, we analyze this solution with respect to the requirements defined in Section 1.

Instance Identification. We refer to an implementation class via its enclosing SPL class (which represents a concrete instance) or indirectly by using a generated factory method (e.g., method `newList()` in Fig. 5), which is part of the generated interface:

```
class MailClient {
  SortListPL sortList = new QuickSortList();
  SortListPL.List createList() {
     return sortList.newList();
  }
}
```

In this example, `createList()` invokes the virtual factory method `newList()` (cf. Fig. 5), which is implemented by an SPL instance that is a subtype of `SortListPL`. This is similar to the use of the `new` operator in virtual classes which simplifies to write generic client code.

In contrast to virtual classes, an SPL instance is represented by a class that implements the interface of a specialized SPL. This means that we can have different implementations of the same (fully) specialized SPL (i.e., that implement the same interface). For example, we can have two sorted lists, one providing a speed optimized implementation and one providing a memory optimized implementation and both can be used polymorphically. It is also possible to implement the same interface in two different SPLs, which allows us to use instances of the SPLs interchangeably. Hence, we extend the interface concept of components to component SPLs. This is different from the virtual class solution (cf. Section 4.2) where we cannot distinguish specialized SPLs from SPL instances.

Subtyping. Subtyping of SPL implementation classes is realized as subtyping between the nested interfaces. As with virtual classes, the interface of a class can thus be used polymorphically. We can use it to reference all variants of a class that are defined in a subytpe of the specialized SPL. For example, a client method

```
void display(SortListPL.List l) { ... }
```

accepts all kinds of sorted lists. Similar to the CaesarJ approach, this solution achieves static subtyping also in case of multiple inheritance between SPL specializations. It is implemented as multiple inheritance between the SPL interfaces and the nested interfaces of implementation classes. For example, a sorted and synchronized list can be a subtype of `SortListPL` and `SyncListPL`. This does not cause problems known from multiple inheritance of implementation classes. Finally, the approach allows us to add new SPL specializations without modifying a client implementation as long as the new variant is not a super type of an existing one (i.e., it does not modify the existing inheritance hierarchy).

The main drawback of this solution is that it does not allow us to statically check whether two objects of implementation classes are compatible with each other (i.e., if they are part of the same SPL instance). For example, using the abstract list interface, we could pass a node of a single linked list to a double linked list causing a runtime type error. With virtual classes, such errors can already be detected by the compiler [9].

Code Reuse. In the presented solution we do not address the problem of code replication. Even though we are using nested interfaces to represent a specialized SPL we cannot use nested classes to represent fragments of SPL implementation classes. It would result in the same problems as observed for mixin composition (entangled feature order and subtyping). Furthermore, it would result in multiple inheritance of implementation classes when multiple inheritance of their interfaces is needed. For example, a sorted and synchronized list would inherit the basic list implementation twice. However, a similar implementation as used in CaesarJ could be used to avoid code replication.

5. DISCUSSION

We presented different approaches to generate component families that allow us to use multiple variants of a component in the same program. In the following, we discuss open issues and suggest how FOP approaches should be changed to provide a viable solution for generating components from an SPL.

5.1 Code Reuse

In CaesarJ, code replication is reduced. It can be completely avoided with an implementation that avoids replication of refinements, e.g., using delegation. This is also possible for generated OO hierarchies and even for the namespace approach. However, it means a more complex code transformation than simply adding a namespace. Furthermore, it may result in an overhead in terms of execution time for introduced indirections, which has to be evaluated.

5.2 When to Use Which Mechanism?

Since all presented solutions have benefits and drawbacks, no mechanism can be generally preferred. We discuss when the different mechanisms should be used.

Plain Static Composition. When using a component, most of the time this means to use a single instance of the component only. This can be accomplished with current approaches for SPL development that use static composition of features. Furthermore, the code transformations used for Jak and FeatureC++ allow us to use generated components in plain Java and C++ clients.

Namespaces. The namespace approach often suffices when a client uses multiple variants of a component. However, it does not support subtyping of generated components. This causes a high effort to write generic code for different component variants. Compared to the advanced solutions that support subtyping, the namespace approach achieves better performance due to the possibility of method inlining. However, a detailed performance evaluation is needed to analyze the actual effect. Due to its simplicity, it can also be used on deeply embedded devices when there is no support for OOP or for some OOP concepts such as virtual methods.

Virtual Classes. Implementing an SPL with virtual classes (e.g., with CaesarJ) allows us to to achieve subtyping of a component hierarchy. By generating virtual classes from an FOP implementation, we avoid the complexity of mixin composition. However, once we have decided for such an SPL implementation, a client developer is faced with the complexity of virtual classes even when not needed. A remaining problem is that we cannot use this solution when the client is developed with a mainstream OO language due to missing support for virtual classes.

Generating SPL Interfaces. To achieve subtyping between SPL specializations in languages that do not support virtual classes, we propose to generate plain OO interfaces to represent specialized SPLs. This allows us to access different variants of a class with the same interface. The approach also allows us to separate SPL instances (implemented as classes) from SPL specializations (implemented as interfaces) and to have different implementations of the same SPL specialization. The main drawback compared to virtual classes is that we loose parts of static type safety on the client side.

5.3 Flexible Feature Composition

To allow programmers to choose the best solution according to the application scenario, we propose to use a flexible approach for feature composition that generates the actually required code. We already support this for plain static composition and generating namespaces / packages. When a component hierarchy and subtyping is needed, we propose to use more advanced approaches: generating virtual classes if supported by the client language or generating a plain OO interface hierarchy otherwise. It is also possible to extend CaesarJ to avoid the problems mentioned above. From CaesarJ code we could then generate code without virtual classes when they are not needed or not supported.

A flexible approach allows us to switch from one implementation mechanism to another by regenerating the components (i.e., when the requirements change). However, this also means that the client, which uses the SPL, has to be changed accordingly. With refactoring feature modules we can automatically refactor the client program as well; but this has to be further analyzed in future work.

6. RELATED WORK

There are also other languages that support virtual classes, which we could have used for our analysis. However, we think that the problems are very similar to those described for CaesarJ.

Nested Intersection. The language J& supports composition of multiple components using nested intersection [17]. It is based on composition of classes and packages with their inner classes similar to virtual classes. J& might be better suited for implementing specialization hierarchies than virtual classes because it defines *static* virtual types, which are attributes of packages or classes and not of objects. However, the composition mechanism does not linearize class extensions, which complicates development of independently composable features. We intend to evaluate the approach for implementing specialization hierarchies in further work.

Mixin Layers. Generics, such as C++ templates, can be used to implement layered designs [22]. Similar to virtual classes, nested classes of a mixin layer extend classes of their super layers. As a precursor of FOP, the language P++ (an extension of C++) provides composition of mixin layers and explicitly defines layer interfaces [2]. Static mixin composition is similar to mixin composition of virtual classes but different instances of a component are generated by template instantiation at compile time. However, as in virtual classes, the feature composition order and subtyping are not independent. Moreover, generating hierarchies with multiple inheritance would result in multiple inheritance of inner classes. Jiazzi solves some of the problems of static mixin composition with concepts similar to virtual classes [16].

Dynamic Feature Composition. Dynamic composition of features means to derive an SPL instance by composing features at runtime (e.g., supported by Delegation Layers [18], Object Teams [10], and FeatureC++). Delegation layers and Object Teams furthermore combine delegation-based composition with virtual classes. Dynamic composition provides more flexibility than static composition of features because the feature selection of an SPL instance is determined in a running program. When this flexibility is needed we do not want to statically define a specialization hierarchy as proposed in this paper.

7. CONCLUSION

Feature-oriented software development lacks support for reusing multiple products of an SPL in the same program. For example, programmers cannot model or implement large software systems that use multiple component variants generated from an SPL. We propose to model and generate *component hierarchies* from a feature-oriented SPL. A component hierarchy allows a programmer to distinguish different variants of a component and provides a subtype relationship between components. This enables client developers to write generic code to be used with different variants of a component.

Based on modeling support for component hierarchies, we apply the concept to feature-based software composition. Since a component hierarchy is only needed when using different variants of the same component, we propose to use a flexible approach to feature composition:

- we use plain static composition if only a single instance of an SPL is used at a time,
- we generate namespaces when using multiple compo-

nent variants at the same time,
- we propose to generate component hierarchies when subtyping is needed: (1) by generating virtual classes or (2) by generating a hierarchy of OO interfaces when the client language does not support virtual classes.

Due to a flexible composition mechanism a developer of a component SPL does not have to consider the special needs of different clients. Based on a feature-oriented implementation of an SPL, a client developer defines the required component hierarchy (or uses a predefined one) and generates the components that correspond to the application scenario and the client language.

In future work, we plan to implement and evaluate the proposed solutions for generating component hierarchies. This means to connect modeling of component hierarchies and feature composition and to extend the FOP code generation process accordingly (e.g., for FeatureC++).

Acknowledgments

We thank Don Batory for discussions about the presented work. Marko Rosenmüller is funded by German Research Foundation (DFG), project number SA 465/34-1.[7] Norbert Siegmund is funded by German Ministry of Education and Research (BMBF), project number 01IM08003C.[8]

8. REFERENCES

[1] I. Aracic, V. Gasiunas, M. Mezini, and K. Ostermann. An Overview of CaesarJ. In *Transactions on Aspect-Oriented Software Development I*, volume 3880 of *LNCS*, pages 135–173. Springer, 2006.

[2] D. Batory, S. Dasari, B. Geraci, V. Singhal, M. Sirkin, and J. Thomas. Achieving reuse with software system generators. *IEEE Software*, pages 89–94, 1995.

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

[4] D. Batory, V. Singhal, M. Sirkin, and J. Thomas. Scalable software libraries. *SIGSOFT Softw. Eng. Notes*, 18(5):191–199, 1993.

[5] G. Bracha and W. R. Cook. Mixin-Based Inheritance. In *Proc. Int'l. Conf. Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA) and the Europ. Conf. Object-Oriented Programming (ECOOP)*, pages 303–311. ACM Press, 1990.

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

[7] K. Czarnecki, S. Helsen, and U. W. Eisenecker. Staged Configuration Using Feature Models. In *Proc. Int'l. Software Product Line Conf. (SPLC)*, volume 3154 of *LNCS*, pages 266–283. Springer, 2004.

[8] E. Ernst. Family Polymorphism. In *Proc. Europ. Conf. Object-Oriented Programming (ECOOP)*, volume 2072 of *LNCS*, pages 303–326. Springer, 2001.

[9] E. Ernst, K. Ostermann, and W. R. Cook. A Virtual Class Calculus. In *Proc. Int'l. Symposium on Principles of Programming Languages (POPL)*, pages 270–282. ACM Press, 2006.

[7]http://fosd.de/multiple
[8]http://vierfores.de

[10] S. Herrmann. Object Teams: Improving Modularity for Crosscutting Collaborations. In *Proc. Int'l. Net.ObjectDays Conf.*, volume 2591 of *LNCS*, pages 248–264. Springer, 2002.

[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-21, Software Engineering Institute, Carnegie Mellon University, 1990.

[12] C. Kästner, S. Apel, and M. Kuhlemann. Granularity in Software Product Lines. In *Proc. Int'l. Conf. Software Engineering (ICSE)*, pages 311–320. ACM Press, 2008.

[13] M. Kuhlemann, D. Batory, and S. Apel. Refactoring Feature Modules. In *Proc. Int'l. Conf. Software Reuse (ICSR)*, pages 106–115. Springer, 2009.

[14] J. Liu and D. Batory. Automatic Remodularization and Optimized Synthesis of Product-Families. In *Proc. Int'l. Conf. Generative Programming and Component Eng. (GPCE)*, pages 379–395. Springer, 2004.

[15] O. L. Madsen and B. Moller-Pedersen. Virtual Classes: A Powerful Mechanism in Object-Oriented Programming. In *Proc. Int'l. Conf. Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, pages 397–406. ACM Press, 1989.

[16] S. McDirmid, M. Flatt, and W. C. Hsieh. Jiazzi: New-Age Components for Old-Fashioned Java. In *Proc. Int'l. Conf. Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, pages 211–222. ACM Press, 2001.

[17] N. Nystrom, X. Qi, and A. C. Myers. J&: Nested Intersection for Scalable Software Composition. In *Proc. Int'l. Conf. Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, pages 21–35. ACM Press, 2006.

[18] K. Ostermann. Dynamically Composable Collaborations with Delegation Layers. In *Proc. Europ. Conf. Object-Oriented Programming (ECOOP)*, volume 2374 of *LNCS*, pages 89–110. Springer, 2002.

[19] C. Prehofer. Feature-Oriented Programming: A Fresh Look at Objects. In *Proc. Europ. Conf. Object-Oriented Programming (ECOOP)*, volume 1241 of *LNCS*, pages 419–443. Springer, 1997.

[20] M. Rosenmüller, M. Kuhlemann, N. Siegmund, and H. Schirmeier. Avoiding Variability of Method Signatures in Software Product Lines: A Case Study. In *Workshop on Aspect-Oriented Product Line Engineering*, pages 20–25, 2007.

[21] M. Rosenmüller and N. Siegmund. Automating the Configuration of Multi Software Product Lines. In *Proc. Int'l. Workshop on Variability Modelling of Software-intensive Systems (VaMoS)*, pages 123–130, 2010.

[22] Y. Smaragdakis and D. Batory. Implementing Layered Designs with Mixin Layers. In *Proc. Europ. Conf. Object-Oriented Programming (ECOOP)*, volume 1445 of *LNCS*, pages 550–570, 1998.

[23] T. Thüm, D. Batory, and C. Kästner. Reasoning about Edits to Feature Models. In *Proc. 31th Int'l. Conf. Software Engineering (ICSE)*, pages 254–264. IEEE CS, 2009.

Language-Independent Reference Checking in Software Product Lines

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

Christian Kästner
Philipps University Marburg, Germany
kaestner@informatik.uni-marburg.de

ABSTRACT

Feature-Oriented Software Development (FOSD) is a paradigm for the development of software product lines. A challenge in FOSD is to guarantee that all software systems of a software product line are correct. Recent work on type checking product lines can provide a guarantee of type correctness without generating all possible systems. We generalize previous results by abstracting from the specifics of particular programming languages. In a first attempt, we present a reference-checking algorithm that performs key tasks of product-line type checking independently of the target programming language. Experiments with two sample product lines written in Java and C are encouraging and give us confidence that this approach is promising.

Categories and Subject Descriptors: D.3.3 [**Software**]: Programming Languages—*Formal Definitions and Theory*; D.3.3 [**Software**]: Programming Languages—*Language Constructs and Features*

General Terms: Languages, Reliability, Design

Keywords: Feature-Oriented Software Development, Software Product Lines, Type Systems, FeatureHouse, FeatureTweezer

1. INTRODUCTION

Feature-Oriented Software Development (FOSD) is a paradigm for the development of software product lines [3,11,13, 26]. The key idea is to modularize software systems in terms of features. A *feature* is a unit of functionality of a software system that satisfies a requirement, represents a design decision, or provides a configuration option [3]. Typically, with a set of features, a developer describes the commonalities and variabilities of a family of software systems of a particular domain (i.e., a software product line).

There are various ways of making the features of a software product line explicit in its document and code base [22]. Compositional approaches such as AHEAD [11] encapsulate the code that belongs to a feature in a cohesive and composable unit. Once we have made features explicit in terms of cohesive and composable units, a customized software system can be generated automatically, simply by selecting a valid subset of features. Commonly, a feature model defines which feature selections are valid [13], and tools are available that check selections for validity [12, 23].

A challenge addressed in recent work is to guarantee that *every* valid feature selection produces a type-correct program [5, 14, 15, 20, 28]. The problem is that, during the life time of a software product line, the feature model and the features' implementations tend to diverge. That is, programs may be well-typed that are not valid in terms of the feature model and programs may be ill-typed that are valid in terms of the feature model. The latter case is problematic because such errors are usually detected only when the program in question is generated based on a feature selection. Due to the possibly very large number of different valid feature selections, this may happen late in the development process, leading to high costs and ripple effects. Generating and compiling all programs is feasible only for small product lines. Product-line type checking addresses this problem by checking the entire code base of a product line once against the product line's feature model to ensure that no valid feature selection produces an erroneous program.

Contemporary product-line–checking approaches are tailored to specific programming languages, development tools, or formalisms, which limits principally their general applicability (see Sec. 5). We would like to explore how far we can implement product-line checking independently of the particular language or formalism used. This approach is inspired by previous work on language-independent feature composition tools [6, 7] and type systems [1]. Naturally, there is a trade-off between generality and expressiveness. That is, in our quest to increase generality, we may lose expressiveness, but we argue that this path is worth exploring and that our initial results are encouraging.

We concentrate on a subclass of possible type errors: *dangling references*. In a product line, code of one feature may refer to code of another feature (e.g., in the form of a method invocation or field access); if the former feature is selected and the latter is not, the former has a dangling reference, reported by the type system. We propose a language-independent model of programs with references, define (two variants of) an algorithm for detecting potential dangling references, and apply our prototypical implementation to two sample product lines written in Java and C.

2. PRELIMINARIES

Our language-independent model of feature-oriented product lines is based on previous work on feature algebra [8] and feature composition tools [6]. For illustration, we use the small example of a variable graph data structure, which is motivated by one of our case studies in Section 4. The graph example consists of the three features GRAPH, WEIGHTS, and DIJKSTRA whose meanings will become clear later. It has been implemented in FeatureHouse, a tool that supports the composition of code written in a number of languages, among them Java [6].

2.1 Feature Structure Trees

We describe the structure of a feature, independently of the programming language, by a *feature structure tree* (FST). An FST organizes the feature's structural elements (e.g., files, classes, fields, or methods) hierarchically. Figure 1 depicts an excerpt of the Java implementation of feature GRAPH and its representation in the form of an FST. One can think of an FST as a stripped-down abstract syntax tree that contains only essential information. The nature of this information depends on the degree of granularity at which software artifacts are to be used, analyzed, or composed, as we discuss below.

```
package graph;
class Graph {
  List<Node> nodes;
  List<Edge> edges;
  void add(Node n) {
    ...
  }
  ...
}
class Node {
  int id; ...
}
class Edge {
  Node src, dest; ...
}
```

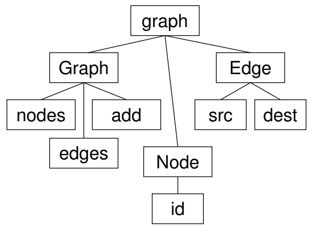


Figure 1: Implementation and FST of feature GRAPH (excerpt).

For example, the FSTs we use to represent Java code contain nodes that represent packages, classes, fields, methods, etc. They do not contain information on the internal structure of methods and so on. A different level of granularity would represent only packages and classes but not methods or fields as FST nodes (coarser granularity). Yet another level of granularity would represent additionally statements or expressions as nodes (finer granularity) [21].

Each node of an FST is labeled with a name and a type (types are not shown in Figure 1, for brevity). We call two nodes *compatible* if they have the same name and type and compatible parents. A node's name[1] corresponds to the name of the structural element it represents and a node's type corresponds to the syntactic category to which the element belongs. For example, class Graph is represented by node Graph of type class (type omitted in Figure 1, for brevity). We must consider both the name and the type to prevent ambiguities.

[1]Depending on the language and purpose, a name could be a simple identifier, a signature, etc.

Note that, at the granularity we chose for Java, the order of nodes could be arbitrary, but this may be different at a finer granularity (e.g., the order of statements matters) and it may also differ for other languages (e.g., the order of C functions and of overlapping Haskell patterns matters) [6]. The FST model has been used to formalize and implement feature composition based on FST superimposition, of which we report elsewhere [6, 8].

It has been shown that the FST model is very general. It can be used with different languages including object-oriented (e.g., Java and C#), imperative (e.g., C), functional (e.g., Haskell), modeling (e.g., UML), and relational languages (e.g., Alloy) [2, 4, 6, 9]. Next, we extend the FST model with references.

2.2 Adding References

First, we introduce the concept of a reference into the FST model. Then, we discuss key design decisions.

References. A product line consists of multiple features, each of which is represented by an FST. Commonly, there are dependences between the individual features. One feature may extend or use another (e.g., in that it invokes a method belonging to the other feature). This kind of reference is common in many languages. Examples of references are field accesses in Java, references between grammar product rules in JavaCC, associations in UML, and so on. Hence, we extend the FST model by references. A reference is a pair of a name of source FST element and a name of a destination FST element. The two elements need not to point to the same FST.

Let us illustrate references by means of our graph example. In Figure 2, we show the implementation and FST of feature WEIGHTS. It refines class Edge of feature GRAPH by adding a new field weight.[2]

```
package graph;
class Edge {
  int weight;
  ...
}
```

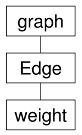


Figure 2: Implementation and FST of feature WEIGHTS (excerpt).

In Figure 3, we show the implementation and FST of feature DIJKSTRA, which implements Dijkstra's algorithm for solving the shortest-path problem. It introduces class Dijkstra that, at some point, accesses field weight of class Edge.

The three features refer to one another: WEIGHTS refers to GRAPH and DIJKSTRA refers to GRAPH and WEIGHTS. A reference has a source (left-hand side) and a target (right-hand side). The source consists of the feature's name and the element's fully-qualified name. The target consists only of the element's fully-qualified name. In the graph example, we

[2]When composing feature GRAPH and feature WEIGHTS, the two declarations of class Edge are merged; this is a form of mixin composition [11].

```
package graph;
class Dijkstra {
  Node[] shortestPath(Node n) {
    ... int w = edge.weight; ...
  }
}
```

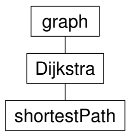


Figure 3: Implementation and FST of feature Dijkstra **(excerpt).**

have the following references, excluding the inner references of a feature to itself:

(Weights, `Edge`) → (`Edge`)
(Dijkstra, `Dijkstra.shortestPath`) → (`Node`)
(Dijkstra, `Dijkstra.shortestPath`) → (`Edge`)
(Dijkstra, `Dijkstra.shortestPath`) → (`Edge.weight`)
...

It is important to note that the target element can be part of different features, so it is not fixed to which element another element points—references are resolved after the desired features have been selected by a user to generate a final program. For example, there may be multiple features that introduce different kinds of weights. It is the task of the reference checker to ensure that there is a proper target for each reference in every valid feature selection.

Discussion. Programs and documents written in various languages can be represented by FSTs [6] and references are a language-independent concept. In fact, the extended FST model represents (a subset of) the context-sensitive abstract grammar of a language, whereas the plain FST model without references represents (a subset of) the context-free abstract grammar. Essentially, the first design decision was to detach the reference model from the underlying language and to base it entirely on FSTs. This way, we attain language independence but may reduce expressiveness in that we cannot represent the full type structure of a language, which may interfere with reference checking. Language independence also implies that references (and FSTs) have to be represented in a general format. For a product line to be checked, FSTs and references have to be extracted by (language-specific) code analysis tools. Then, reference checking is generic and uniform.

A second design decision we made is that we model references as *pairs* of FST elements. This is the simplest model possible and we use it until we encounter the need for a more complex model. In some languages, we may need references with multiple possible targets. Furthermore, some languages may need a more dynamic view of references, for instance, to take dynamic binding into account.

The third design decision we made is that a reference contains, beside the source and target elements, also the source feature; in contrast, the target feature is not contained in the reference and undefined until the user selects a set of features. The rationale is that, if a piece of code contains a reference, we know to which feature the piece belongs, but we do not know which feature provides a proper target for the reference—in fact, there may be multiple features. However, in some languages, there may be situations in which the type of the target element is relevant (e.g., a field with type String instead of int). Since we currently do not support full typing, we cannot address this issue without losing language independence.

FSTs and their references provide insight into the structural interactions between features. The knowledge about the features of a product line, their references, and their valid selections (i.e., the feature model) allows us to formulate a language-independent reference-checking algorithm.

2.3 Feature Models

Before we describe the reference-checking algorithm, we repeat briefly the basics of feature models. A *feature model* describes the valid feature selections of a software product line [19]. There are different approaches and notations for describing feature models [12]. We use the approach of Batory in which a feature model is represented by a propositional formula [10]. The formula contains, for each feature, a boolean variable and expresses the constraints between features. Most other notations can be translated to propositional formulas.

A propositional formula describing the variability of our graph example could look as follows:

$$(\textsc{Weights} \lor \textsc{Dijkstra}) \Rightarrow \textsc{Graph} \tag{1}$$

The formula states that, whenever Weights or Dijkstra are selected, then also Graph has to be selected. A consequence is that the features Weights and Dijkstra are optional and independent (which can lead to a dangling reference, as we will explain shortly).

Solver technology can be employed to answer a number of questions on feature models including whether a feature selection is valid or whether a feature is always, sometimes, or never present when another feature is present [12], which is interesting information for reference checking. For example, feature Graph is *always* present when one of the other two features is present and feature Weights is *sometimes* present when feature Dijkstra is present.

Typically, a feature model describes the variability of a product line without considering a particular implementation. Hence, the variability of a domain does not necessarily need to be consistent with the variability of the corresponding implementation. There may be valid feature selections that lead to programs with type errors. For example, assuming the feature model of Equation 1, there is a valid feature selection that leads to an incorrect program in our graph example. By selecting Graph and Dijkstra only, we get a type error because Dijkstra refers to field weight, which belongs to the non-selected feature Weights.

The graph example illustrates that implementation variability may differ from domain variability. Both kinds of variability can be described by feature models. The feature model that describes the implementation variability of our graph example, henceforth called the *implementation model*, is:

$$\begin{aligned}(\textsc{Weights} \lor \textsc{Dijkstra}) \Rightarrow \textsc{Graph} \quad \land \\ (\textsc{Dijkstra} \Rightarrow \textsc{Weights})\end{aligned} \tag{2}$$

Compared to the feature model of Equation 1, henceforth called the *domain model*, it contains an additional constraint that states that, whenever Dijkstra is selected, Weights has to be selected, too. A key task of reference checking is to derive information on implementation variability and to check it against domain variability to discover potential dangling references.

Algorithm 1 Global reference-checking algorithm.

```
Require: $\mathbb{F}$ := set of feature names
Require: $FT$ := FST table
Require: $RT$ := reference table
Require: $DM$ := domain feature model
 1: // derive formula of implementation feature model
 2: $IM := \bigwedge_{((f,src),tgt)\in RT}(f \Rightarrow \bigvee_{f_i\in\mathbb{F},tgt\in FT(f_i)} f_i)$
 3: // check whether the two models are consistent
 4: $sln := \mathsf{solve}(DM \Rightarrow IM)$
 5: if $sln \neq true$ then
 6:     // determine counterexample
 7:     $cex := \mathsf{counter}(DM \Rightarrow IM)$
 8:     $\mathsf{print}(cex)$
 9:     // identify dangling references
10:     $dref := \{(ref = ((f, src), tgt)) \mid$
11:         $ref \in RT, f \in cex, (\nexists f \in cex : tgt \in FT(f))\}$
12:     for all $(ref = ((f,src), tgt)) \in dref$ do
13:         // identify features with potential targets
14:         $ptgt := \{f \mid f \in (\mathbb{F}\backslash cex),\ tgt \in FT(f)\}$
15:         $\mathsf{print}(ref)$
16:         $\mathsf{print}(ptgt)$
17:     end for
18: end if
```

Algorithm 2 Local reference-checking algorithm.

```
Require: $\mathbb{F}$ := set of feature names
Require: $FT$ := FST table
Require: $RT$ := reference table
Require: $DM$ := domain feature model
 1: for all $(ref = ((f,src), tgt)) \in RT$ do
 2:     // derive formula of reference
 3:     $RM := (f \Rightarrow \bigvee_{f_i\in\mathbb{F},tgt\in FT(f_i)} f_i)$
 4:     // check for consistency
 5:     $sln := \mathsf{solve}(DM \Rightarrow RM)$
 6:     if $sln \neq true$ then
 7:         // determine counterexample
 8:         $cex := \mathsf{counter}(DM \Rightarrow RM)$
 9:         $\mathsf{print}(cex)$
10:         // identify features with potential targets
11:         $ptgt := \{f \mid f \in (\mathbb{F}\backslash cex),\ tgt \in FT(f)\}$
12:         $\mathsf{print}(ref)$
13:         $\mathsf{print}(ptgt)$
14:     end if
15: end for
```

3. REFERENCE-CHECKING ALGORITHM

There are two variants of our reference-checking algorithm, each of which has been inspired by a different branch of previous work (see Sec. 5): the *global reference-checking algorithm* creates a single propositional formula (i.e., the implementation feature model) that covers all references [14, 15, 28]; the *local reference-checking algorithm* creates a propositional formula for each reference that covers exactly the constraints implied by this reference [5, 20].

Global Reference-Checking Algorithm. In Algorithm 1, we list the global variant of the reference-checking algorithm including comments. It takes as input the following ingredients of a product line: the domain feature model, the FSTs of all features, and the references between FST elements. Based on this information, the global variant of the algorithm proceeds in three steps:

1. The input FSTs and references are analyzed to obtain the implementation feature model. To this end, for each reference, all proper target elements are searched. At least one target (i.e., its feature) must be present when the source feature is present. This requirement is added as a disjunctive clause to the propositional formula of the implementation model (Line 2).
2. A SAT solver checks whether the implementation model is consistent with the domain model (Line 4).
3. If the two models are not consistent, a counterexample is generated, which is a set of features that contains dangling references (Line 7). Based on the counterexample, for each dangling reference, all features with proper targets are identified (Lines 12–17).

Local Reference-Checking Algorithm. In Algorithm 2, we list the local variant of the reference-checking algorithm, including comments. Like in the global variant, it takes as input the domain feature model, the FSTs of all features, and the references between FST elements. The algorithm proceeds in three steps:

1. Rather than creating a single propositional formula for the entire implementation model, one propositional formula per reference is generated, which describes the constraints implied by the reference, called the *reference model*. Again, at least one target of the reference (i.e., its feature) must be present when the source feature is present (Line 3).
2. A SAT solver checks whether the constraints imposed by each single reference are consistent with the domain model (Line 5).
3. If the constraints of some reference are not satisfied, a counterexample is generated (Line 8). Based on the counterexample, for the dangling reference in question, all features with proper targets are identified (Line 11).

```
erroneous feature selection:
  [Graph,Dijkstra]
dangling reference:
  (Dijkstra.shortestPath, Edge.weight)
  in feature 'Dijkstra'
features that provide proper targets:
  [Weights]
```

Figure 4: Output of FEATURETWEEZER when checking the graph example.

Discussion. Both variants of the reference-checking algorithm have a similar input-output behavior. They expect domain and structural information and provide information on dangling references and potential target features. For our graph example, both variants of the algorithm would produce an output like the one shown in Figure 4.

So what is the difference between the two variants and why have researchers invented them in the first place? A key difference is the size and number of propositional formulas to be checked for consistency with the domain model. In the global variant, we have a single, possibly large formula; in the local variant, we have many smaller formulas. This difference may be crucial for performance, an issue that has gained too little attention in the past. Admittedly, there is initial evidence that extracting a complete implementation model is possible in linear time [27] and that solving large formulas that represent feature models is possible in practice in polynomial time [25]. But there is also evidence that solving many small formulas is efficient because intermediate results can be cached and reused [5]. In the future, we intend to address this issue systematically. A contribution of our language-independent model (and tool) is that we can represent both variants at an abstract level, reveal their principal differences, and provide a basis for experiments.

Another difference between the local and global variant of the reference-checking algorithm is error reporting. The local variant is finer-grained in that it identifies potential dangling references directly and points to the corresponding locations in the code. The global variant searches first for an erroneous feature selection and identifies then potential dangling references, but only for this selection. The local variant identifies all potential dangling references. This makes the debugging process less iterative and more efficient.

4. PROTOTYPE AND CASE STUDIES

As a proof of concept, we have been developing a prototype of a product-line reference checker in Haskell, called FeatureTweezer.[3] Checking for dangling references in a product line, FeatureTweezer expects the product line's FSTs, the references, and the feature model. Currently, an FST is encoded as a set of prefix-closed identifiers (the prefix encodes the path in the FST), each of which denotes an FST element. A reference is encoded as a pair of FST element identifiers. A feature model is encoded in the GUIDSL format [10], but, as illustrated in Figure 5, alternative formats are possible. We depict the input data for our graph example to FeatureTweezer in Figure 6, simplified and adapted for presentation purposes.

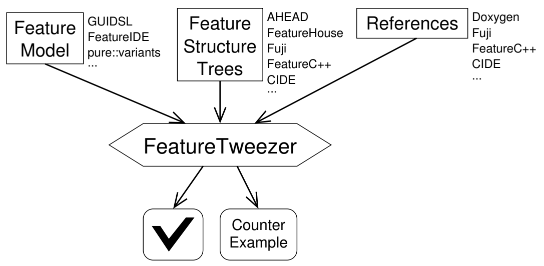


Figure 5: Input and output of FeatureTweezer.

FeatureTweezer processes all input information, transforms it into a suitable internal format, and feeds it into the reference-checking algorithm (see Sec. 3). To check reference constraints and implementation models against the domain model, we use *funsat*[4], an open-source native Haskell SAT solver. It provides a counterexample if the models are not consistent. This information is used to find sources of dangling references and potential features that provide proper targets.

To gather experience with FeatureTweezer, we applied it to two sample product lines: the graph product line (GPL)

```
                                         domain feature model
GraphExample : Graph [Weights] [Dijkstra] ;
```

```
                   feature structure trees (type feature element)
ClassDecl Graph Graph
FieldDecl Graph Graph.nodes
FieldDecl Graph Graph.edges
...
ClassDecl Weights Edge
FieldDecl Weights Edge.weight
...
ClassDecl Dijkstra Dijkstra
MethodDecl Dijkstra shortestPath
```

```
                     references (type (feature element) element)
ClassRef (Weight Edge) Edge
ClassRef (Dijkstra Dijkstra.shortestPath) Node
ClassRef (Dijkstra Dijkstra.shortestPath) Edge
MethodRef (Dijkstra Dijkstra.shortestPath) Edge.weight
...
```

Figure 6: Input data of the graph example for FeatureTweezer (simplified).

of Lopez-Herrejon and Batory [24] and the feature-oriented email client of Hall [18]. The former product line is implemented in Java and the latter is implemented in C. A key challenge is to extract the necessary information for FeatureTweezer (FSTs, references, and feature model). For the purpose of our initial experiments, we chose a pragmatic approach. We extracted the FSTs with FeatureHouse[5] and the references with Doxygen[6] and CCVisu[7]. We extended FeatureHouse to export the FSTs of a product line to the expected format and we configured Doxygen and CCVisu such that the element identifiers in the references correspond to the element identifiers in the FSTs; the feature models were available as part of the sample product lines.

The fact that we were able to check two product lines written in two different languages illustrates the potential of our approach. We did not find bugs in the two sample product lines. This is not surprising because they are rather small and well-tested. However, the product lines were useful for testing our tool by introducing errors artificially. The size of the sample product lines do not allow us to draw any conclusions on the differences in performance and resource consumption between the local and the global reference-checking algorithm. In further work, we intend to check more and larger product lines, written in different languages, to discover real bugs and to measure performance and resource consumption.

5. RELATED WORK

Our work on reference checking feature-oriented product lines has been motivated by previous work on type systems for product lines [5, 14, 15, 20, 28]. Reference checking is an important subset of type checking, so we believe we have taken an important step toward a general tool suite for product-line engineering.

Thaker et al. developed a type system for feature-oriented product lines, based on Java, that does not check all in-

[3]FeatureTweezer, including the examples and case studies, is available on the Web: `http://www.fosd.de/FT/`

[4]`http://github.com/dbueno/funsat/`

[5]`http://www.fosd.de/fh/`

[6]`http://www.doxygen.org`

[7]`http://www.sosy-lab.org/~dbeyer/CCVisu/`

dividual programs but the individual feature implementations [28]. In a number of case studies, they found numerous hidden errors using their type rules. Our global variant of the reference-checking algorithm is inspired by their type-checking algorithm, which generates a single, large propositional formula representing implementation variability. Furthermore, Delaware et al. developed a formal model of the type system of Thaker et al. and proved its soundness [15].

Even previously to the work of Thaker et al., Czarnecki and Pietroszek presented an automatic verification procedure for ensuring that no ill-formed UML model template instances will be generated from a valid feature selection [14], which also uses a global reference-checking algorithm. That is, they type check product lines that consist not of Java programs but of annotated UML models (they use OCL constraints to express and implement a kind of type system for UML; annotations denote features).

Our local reference-checking algorithm is inspired by our own work on formal type systems for product lines [5, 20]. Kästner and Apel have developed the formal calculus CFJ based on a subset of Java and a set of type rules for checking annotation-based product lines [20]. Like in the approach of Czarnecki and Pietroszek, and in contrast to AHEAD, variability is implemented with #ifdef-like directives or similar annotations on the source code [21]. Kästner and Apel use a local type-checking algorithm to guarantee type correctness. Similarly, Apel et al. [5] define a formal model of a feature-oriented language and a corresponding product-line type system based on an compositional approach and a subset of Java.

All of the approaches we discussed so far are tailored to specific languages and core languages. An interesting aspect is that our model is able to express reference checking in both composition-based and annotation-based product lines. Both can be reduced to FSTs and references.

Tartler et al. demonstrate that implementation models of C code including preprocessor directives can be extracted in linear time [27]. As in the global reference-checking algorithm, they extract the implementation model once in the form of a single propositional formula. Mendonca et al. demonstrate that consistency checking in the global approach is possible in practice in polynomial time [25]. Apel et al. illustrate how caching can be used to scale the local variant of reference checking by reusing intermediate results [5]. These pieces of work illustrate that there is a potential for tuning the performance of type checking product lines. Our approach and tool can provide a basis for further experiments in this direction.

6. CONCLUSION

We presented a language-independent reference checking algorithm for feature-oriented product lines. To this end, we extended the existing model of feature structure trees with references. Our algorithm checks an entire product line and reports whether any valid feature selection results in a program that contains a dangling reference. The algorithm is based on FSTs, extended with references, and the feature model of a product line. We developed a prototypical tool called FeatureTweezer, which we used to check two sample product lines written in Java and C for dangling references. We believe that our work is a first step toward a more general understanding of and more general tools for feature-oriented product lines implemented using both compositional units and annotations.

In further work, we plan to extend our approach based on language-independent, cross-language, or extensible type systems [1,16,17] with other well-formedness criteria such as mutual exclusion, typing, and subtyping. Furthermore, we plan to extend existing feature algebraic models with references and to reason about the effects of references on algebraic properties. Finally, we intend to conduct further case studies of different domains, written in different languages and provide evidence on the generality of our approach, to discover real bugs, and to measure performance and resource consumption of local and global reference checking.

Acknowledgments

Wolfgang Scholz is supported by the German Research Foundation (DFG—AP 206/2-1). Kästner's work is supported by the European Research Council (ERC #203099).

7. REFERENCES

[1] S. Apel and D. Hutchins. A Calculus for Uniform Feature Composition. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 32(5):Article 19, 2010.

[2] S. Apel, F. Janda, S. Trujillo, and C. Kästner. Model Superimposition in Software Product Lines. In *Proceedings of the International Conference on Model Transformation (ICMT)*, volume 5563 of *LNCS*, pages 4–19. Springer-Verlag, 2009.

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

[4] S. Apel, C. Kästner, A. Größlinger, and C. Lengauer. Feature (De)composition in Functional Programming. In *Proceedings of the International Conference on Software Composition (SoftComp)*, volume 5634 of *LNCS*, pages 9–26. Springer-Verlag, 2009.

[5] S. Apel, C. Kästner, A. Größlinger, and C. Lengauer. Type Safety for Feature-Oriented Product Lines. *Automated Software Engineering – An International Journal (2010)*, 17(3):251–300, 2010.

[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 CS, 2009.

[7] S. Apel and C. Lengauer. Superimposition: A Language-Independent Approach to Software Composition. In *Proceedings of the International Symposium on Software Composition (SoftComp)*, volume 4954 of *LNCS*, pages 20–35. Springer-Verlag, 2008.

[8] S. Apel, C. Lengauer, B. Möller, and C. Kästner. An Algebraic Foundation for Automatic Feature-Based Program Synthesis. *Science of Computer Programming (SCP)*, 75(11):1022–1047, 2010.

[9] S. Apel, W. Scholz, C. Lengauer, and C. Kästner. Detecting Dependences and Interactions in Feature-Oriented Design. In *Proceedings of the International Symposium on Software Reliability Engineering (ISSRE)*. IEEE CS, 2010.

[10] D. Batory. Feature Models, Grammars, and Propositional Formulas. In *Proceedings of the*

International Software Product Line Conference (SPLC), volume 3714 of *LNCS*, pages 7–20. Springer-Verlag, 2005.

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

[12] D. Benavides, S. Segura, and A. Ruiz-Cortes. Automated Analysis of Feature Models 20 years Later: A Literature Review. *Information Systems*, 35(6):615–636, 2010.

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

[14] K. Czarnecki and K. Pietroszek. Verifying Feature-Based Model Templates Against Well-Formedness OCL Constraints. In *Proceedings of the International Conference on Generative Programming and Component Engineering (GPCE)*, pages 211–220. ACM Press, 2006.

[15] B. Delaware, W. Cook, and D. Batory. Fitting the Pieces Together: A Machine-Checked Model of Safe Composition. In *Proceedings of the International Symposium on Foundations of Software Engineering (FSE)*, pages 243–252. ACM Press, 2009.

[16] M. Grechanik, D. Batory, and D. Dewayne. Design of Large-Scale Polylingual Systems. In *Proceedings of the International Conference on Software Engineering (ICSE)*, pages 357–366. IEEE CS, 2004.

[17] N. Haldiman, M. Denker, and O. Nierstrasz. Practical, Pluggable Types for a Dynamic Language. *Computer Languages, Systems and Structures*, 35(1):48–62, 2009.

[18] R. Hall. Fundamental Nonmodularity in Electronic Mail. *Automated Software Engineering*, 12(1):41–79, 2005.

[19] 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, SEI, CMU, 1990.

[20] C. Kästner and S. Apel. Type-Checking Software Product Lines – A Formal Approach. In *Proceedings of the International Conference on Automated Software Engineering (ASE)*, pages 258–267. IEEE CS, 2008.

[21] C. Kästner, S. Apel, and M. Kuhlemann. Granularity in Software Product Lines. In *Proceedings of the International Conference on Software Engineering (ICSE)*, pages 311–320. ACM Press, 2008.

[22] C. Kästner, S. Apel, and M. Kuhlemann. A Model of Refactoring Physically and Virtually Separated Features. In *Proceedings of the International Conference on Generative Programming and Component Engineering (GPCE)*, pages 157–166. ACM Press, 2009.

[23] C. Kästner, T. Thüm, G. Saake, J. Feigenspan, T. Leich, F. Wielgorz, and S. Apel. FeatureIDE: Tool Framework for Feature-Oriented Software Development. In *Proceedings of the International Conference on Software Engineering (ICSE)*, pages 611–614. IEEE CS, 2009.

[24] R. Lopez-Herrejon and D. Batory. A Standard Problem for Evaluating Product-Line Methodologies. In *Proceedings of the International Conference on Generative and Component-Based Software Engineering (GCSE)*, volume 2186 of *LNCS*, pages 10–24. Springer-Verlag, 2001.

[25] M. Mendonca, A. Wasowski, and K. Czarnecki. SAT-based Analysis of Feature Models is Easy. In *Proceedings of the International Software Product Line Conference (SPLC)*, pages 231–240. SEI, CMU, 2009.

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

[27] R. Tartler, J. Sincero, D. Lohmann, and W. Schröder-Preikschat. Efficient Extraction and Analysis of Preprocessor-Based Variability. In *Proceedings of the International Conference on Generative Programming and Component Engineering (GPCE).* ACM Press, 2010.

[28] S. Thaker, D. Batory, D. Kitchin, and W. Cook. Safe Composition of Product Lines. In *Proceedings of the International Conference on Generative Programming and Component Engineering (GPCE)*, pages 95–104. ACM Press, 2007.

Raising family is a good practice

Vinay Kulkarni
Tata Consultancy Services
54B, Industrial Estate, Hadapsar
Pune, 411013 INDIA
+91 20 66086301
vinay.vkulkarni@tcs.com

ABSTRACT

The need for adaptiveness of business applications is on the rise with continued increase in business dynamics. Code-centric techniques show unacceptable responsiveness in this dynamic context as business applications are subjected to changes along multiple dimensions that continue to evolve simultaneously. Recent literature suggests the use of product line architectures to increase adaptiveness by capturing commonality and variability to suitably configure the application. Use of model driven techniques for developing business applications is argued as a preferable option because platform independent specification can be retargeted to technology platform of choice through a code generation process. Business applications can be visualized to vary along five dimensions, namely, Functionality (F), Business process (P), Design decisions (D), Architecture (A) and Technology platform (T). Use of models is largely limited to F and P dimensions in commonly used model-driven development techniques thus limiting the benefits of product line concept to these two dimensions. We argue this is not sufficient to achieve the desired adaptiveness, and it is critical to extend the product line concept to D, A and T dimensions also. To address adaptation needs of business applications, this paper presents a model-driven generative approach that further builds on the ideas of separation of concerns, variability management and feature modeling. Early experience and lessons learnt are discussed, and future work outlined.

Categories and Subject Descriptors

D.2.m [**Software Engineering**]: Miscellaneous – *reusable software*.

General Terms

Management, Economics, Human Factors, Standardization, Languages

Keywords

Commonality, Variability, Adaptiveness, Model-driven development, Business applications, Product lines, Product families

1. INTRODUCTION

Rapid evolutions of technology platforms and business demands have contributed to significant increase in business dynamics in recent years. The increased dynamics put new requirement on businesses while opening up new opportunities that need to be addressed in an ever-shrinking time window. Stability and robustness seem to be giving way to agility and adaptiveness. This calls for a whole new perspective for designing (and implementing) software-intensive systems so as to impart these critical properties. Traditional business applications typically end up hard-coding the operating context in their implementation. As a result, adaptation to a change in its operating environment leads to opening up of application implementation resulting in unacceptable responsiveness.
Typical database-intensive enterprise applications are realized conforming to distributed architecture paradigm that requires diverse set of technology platforms to implement. Such applications can be visualized along five dimensions, namely, Functionality (F), Business process (P), Design decisions (D), Architecture (A) and Technology platform (T). A purpose-specific implementation makes a set of choices along these dimensions, and encodes these choices within application implementation in a scattered and tangled manner. This is an expensive and error prone process demanding large teams with broad-ranging expertise in business domain, architecture and technology platforms. Large size of an enterprise application further exacerbates this problem. Model-driven development alleviates this problem to an extent by automatically deriving an implementation from its high-level specification using set of code generators [20]. However, the scattering and tangling is the principal obstacle in agile adaptation of existing implementation for the desired change. Product line architectures aim to increase adaptiveness by capturing commonality and variability to enable application configurability. As the use of models is limited to F and P dimensions in commonly seen model-driven development techniques, the benefits of product line concept are also limited to these two dimensions. Therefore, it is critical to extend the product line concept to D, A and T dimensions also.

We present a model-driven approach that addresses adaptations needs along all the five dimensions using a specification-driven generative approach [8]. F and P dimension meta models are extended to support modeling of variability that is specified using feature model techniques. A meta model connecting these models

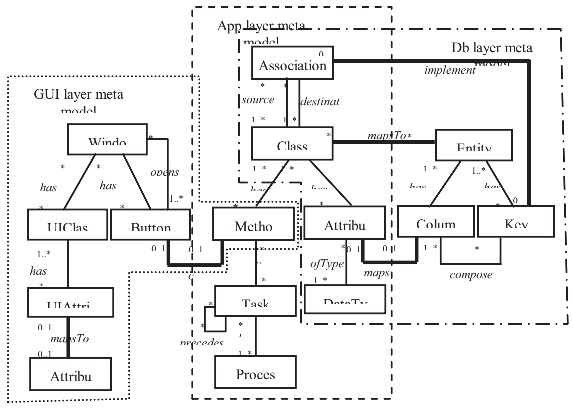


Fig. 1 Unified meta model for business application

to the features in the feature model is presented. Building block abstraction as a means to implement a D or A or T feature is presented. A MDD code generator can be modeled as a hierarchical composition of building blocks, and maps easily to a consistent well-formed configuration of a feature model along D, A and T dimensions.

Section 2 describes model-based techniques we developed to automate development of business applications, and discusses our experience in using this approach to develop several large business applications on a variety of technology platforms and architectures. Section 3 presents an abstraction for organizing model-based code generators as a hierarchical composition of reusable building blocks, and discusses our experience and lessons learnt. Section 4 describes extensions to application specifications so as to model an application family. Section 5 discusses some of the related work. Section 6 concludes with a brief summary of early use of the proposed approach, and outlines future work.

2. GENERATING BUSINESS APPLICATIONS FROM MODELS

A typical database-intensive business application can be seen as a set of services with each service delivering specific business intent. These applications are best implemented using a layered architecture paradigm wherein each layer encapsulates a specific kind of data manipulations e.g. database access layer implements functionality such as primary-key based create/modify/get/delete and complex data accesses like joins, user interface layer implements how the data should be displayed on a screen and how the user will interact, etc. Thus, a set of code patterns recur in the implementation of an architectural layer. An architectural layer interfaces with its adjoining layer through a priori well-defined protocol. Thus, execution of a business application can be seen as an assembly-line of architectural layer specific processors that manipulate the work item, i.e. data corresponding to input and output parameters of a service, in a pre-defined manner before passing it over to the next processor in the assembly-line. As the processing is data-centric and a priori known, it can be easily generated for a given data definition.

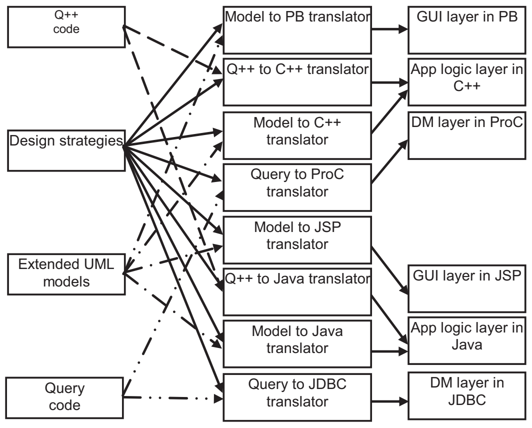


Fig. 2 Scattering and tangling in code generation

Model-driven development approach starts with definition of an abstract specification that is to be transformed into a concrete implementation on a given target architecture [18]. The target architecture is usually layered with each layer representing one view of the system e.g. Graphical User Interface (GUI) layer, application logic layer and database layer. The modeling approach constructs the application specification using different abstract views- each defining a set of properties corresponding to the layer it models. We decompose an application specification into three such models- GUI layer model, Application layer model and Db layer model as shown in Fig. 1. We consider three meta models, namely GUI layer meta model, Application layer meta model and Db layer meta model, for the three view specifications. Each models views of a single Unified meta model as depicted in Fig. 1. Having a single meta model allows to specify integrity constraints to be satisfied by the instances of related model elements within and across different layers. This enables independent transformation of GUI layer model, Application layer model, and DB layer model into their corresponding implementations namely GUI layer code, Application layer code and Db layer code with assurance of integration of these code fragments into a consistent whole. These transformations are performed using code generators. The transformations are specified at meta model level and hence are applicable for all its model instances. If each individual transformation implements the corresponding specification and its relationships with other specifications correctly then the resulting implementations will glue together giving a consistent implementation of the specification.

2.1 Experience and lessons learnt

UML [16] modeling helped in early detection of errors in application development cycle. We associated with every model a set of rules and constraints that defined validity of its instances. These rules and constraints included rules for type checking and for consistency between specifications of different layers. We kept the models independent of implementation technology so as to be able to retarget them to multiple technology platforms i.e. gui platform, middleware, programming language, rdbms and operating system. We defined a higher level domain-specific language to specify business logic [10]. Non-primary key based

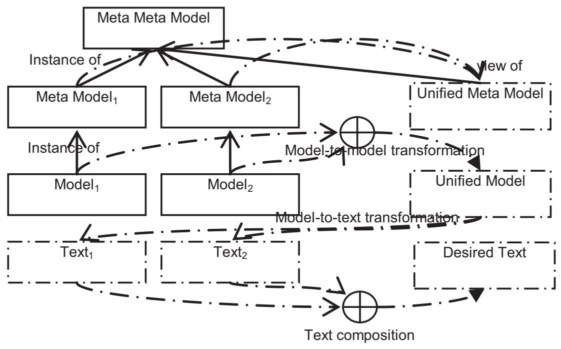


Fig. 3. Model-driven code generation architecture

complex data accesses were specified using a SQL-variant. Application specified in terms of models, Q++ and SQL-variant language was transformed into the target technology platforms encoding the chosen design strategies and architectural choices through a set of code generators. We preserved the divide and conquer strategy by having a code generator each for architectural layers.

Automated code generation resulted in significantly higher productivity in terms of lines of code [22]. Moreover, encoding of design strategies, guidelines and best practices into the code generators resulted in uniformly high code quality. Generation of interface code between the various architectural layers ensured smooth integration of independently generated code artifacts. We discovered that design strategies and architectural choices for no two applications were exactly alike necessitating development of application-specific code generators. Moreover, many architectural and design strategies cut across the layers. This required each tool to be aware of these cross cutting aspects. As a result, customizing for such cross cutting aspects required consistent modifications to several tools leading to maintenance problems. Increased acceptance of the approach led to the ironical situation wherein productivity toolset team became bottleneck for application delivery [21].

3. ORGANIZING MULTIPLE TOOLSETS INTO A FAMILY

As can be seen from figure 2, different code generators are needed to deliver the same business functionality on different technology platforms. This is despite these platform-specific code generators sharing a great deal of common functionality and mostly differing only in the use of primitives offered by the target technology platform e.g. syntax differences of programming languages, data type differences of databases, etc. Even while delivering identical business functionality on identical technology platforms, need to deliver onto different architectures e.g. synchronous, queue-based messaging etc demands different code generators. Similarly, use of different design strategies demands different code generators.

Thus, domain of model-based code generators can be described as a feature diagram [9] where intermediate nodes denote the variation points along D, A and T dimensions; the leaf nodes denote choices available for each variation point i.e. variations; and dependency between variation points expressed in terms of conditional expressions over their respective variations. The desired model-based code generator is a valid configuration over the feature diagram. However, feature diagram is just a declarative specification in terms of labels, and needs support for tracing a feature (i.e. the label) to its implementation. Recommended way for implementing the desired feature configuration is an ordered composition of the implementations of the set of constituent features. But, strict order is not always possible for inter-dependent features.

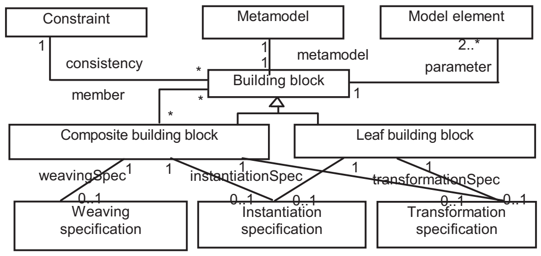


Fig. 4. Building block meta model

The tangling of model-based code generators, as shown in figure 2, is due to lack of separation of the various concerns, namely, technology platform, architecture and design strategies, and the cross-cutting nature of design strategies. An improved architecture for model-based code generation is where the models are successively refined through application of the various design strategies to a stage from where a platform specific implementation can be realized through a simple task of model-to-text transformation. As the platform specific code generators are independent of design strategy related issues, the same model-to-text transformation specifications can be reused with different design strategies and vice versa. This separation of concerns enables a tool variant to be viewed as a composition of design strategy and technology platform aspects of choice.

3.1 Building block

Building block is an abstraction that provides a traceable path to implementation for a feature as per the generic model-driven code generation architecture as shown in fig. 3. A building block is localized specification of a concern in terms of concern-specific meta model, model to model transformation, and model to text transformation. Building blocks are composable, enabling a model-driven code generator to be organized as a composition of a set of reusable building blocks, each encapsulating a specific concern. Figure 4 shows the building block meta model. A building block is essentially a means for expressing how a concern specification is transformed into models and code.

A model-driven code generator is specified as a hierarchical composition of building blocks of interest. Building blocks are of two kinds: *leaf building block* and *composite building block*. The instantiation specification of a leaf building block specifies how to stamp out the model elements of the unified model from the concern-specific model and the transformation specification captures how the model is transformed to code. We use QVT language [13] to specify the former and Mof2Text language [12] to specify the latter. A composite building block specifies how to compose its child building blocks. Weaving specification of a composite building block specifies how the code generated by its member building blocks is woven together. The process of model-driven code generation is realized through a post-order traversal

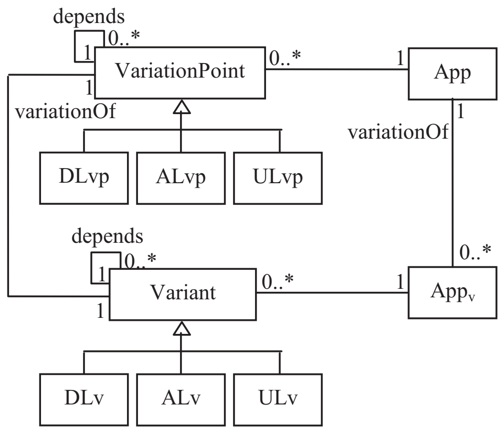


Fig. 5. A generic variability meta model

of the building block hierarchy in three sequential steps, namely, *Instantiation*, *Transformation* and *Weaving*. The instantiation step stamps out models and merges them. The transformation step transforms models into code snippets and generates weaving specifications for composing them. The weaving step composes the generated code snippets by processing the weaving specifications.

We translate a model (M_u) that is an instance of a unified meta model (MM_u) to various software artifacts like Java code, JDBC code, JSP code and a variety of configuration specifications in XML. Limiting aspect weaving only to code level artifacts would necessitate specialized weavers for Java, JDBC, JSP, XML etc. each having separate join point models. Also, this approach would necessitate some commonality over these join point models so as to have an integrated Java application. With increased number of software artifacts to be produced the approach becomes increasingly complex as essentially it amounts to building aspect infrastructure for each such artifact. We address this problem by specifying aspect weaving at the unified meta model level and performing it at the model level whenever possible. Unified meta model enables specification of relationships between the various (sub) modeling languages. A reflexive meta modeling framework provides the necessary infrastructure to define and integrate the various modeling languages of interest and a meta model aware model transformation framework provides the necessary technology to address model weaving requirements [10]. Performing aspect weaving at the model level also, whenever possible, results in reuse of model based code generators such as model-to-Java, model-to-JDBC, model-to-JSP and model-to-XML as these code generators are specified at the unified meta model level.

Multiple variants of a code generator realized as different compositions of building blocks can be easily organized into a family as follows,

- Commonality across variants can be specified using a set of common building blocks
- Functionality specific to each variant can be specified using a set of variant building blocks
- Composable nature of building blocks enables realization of the desired family member as a composition of suitable common and variant building blocks

Thus, building block abstraction provides a traceable path for a feature towards its implementation.

3.2 Experience and lessons learnt

Use of building block abstraction has enabled our toolset to be organized as a family or a product line wherein a tool variant can be easily composed from design strategy and technology platform aspects of choice. Containment of change impact due to localization and increased reuse due to composability have led to quick turnaround time for delivering a tool variant [17]. Use of a higher-level model-aware transformation language has made maintenance and evolution of the product line easy [14]. Also, building block abstraction has enabled us to organize the development team along two independent streams, namely, technology platform experts and design experts.

4. MODELING APPLICATION FAMILIES

Our organization discovered that solutions being delivered to different players in the same business domain were not exactly alike even for identical business intent. With toolset providing no means to capture commonality and variability, application development teams had to resort to copy-paste. As a result, what should have been a variant of an application ended up being a separate application thus leading to maintenance and evolution problems. These problems compounded with every new solution being delivered.

The approach described in section 2 generated a layered application implementation from a similarly layered specification. Each layer of application specification is an instance of its specific meta model. Now we describe how each of these meta models is enhanced to support the family concept [3]. Figure 5 depicts a meta model for capturing variability in a generic way. An application is viewed as a set of a priori defined *variation points* that could possibly be inter-dependent. Possible *variants* for each variation point are identified. Inter-dependence of variation points translates to similar relationship between their variants. Since our interest is database-intensive business applications that are typically implemented using a layered architecture, we identify variation points for each architectural layer. For instance, *DLvp* denotes variation points in database access layer, *ALvp* denotes variation points in application layer, and *ULvp* denotes variation points in user interface layer. Similarly, *DLv*, *ALv* and *ULv* denote variants in database access, application, and graphical user interface layers respectively. A set of *DLv* that honour dependency constraints between *DLvp*, a set of *ALv* that honour dependency constraints between *ALvp*, and a set of *ULv* that honour dependency constraints between *ULvp* constitute a complete, well-formed and consistent application variant App_v.

4.1 Approach

4.1.1 Application layer

Application layer specifies the business logic in terms of *Class*, *Attribute* and *Operations*. Being an encapsulation of both structural and behavioral aspects, Class is the natural choice for supporting the family concept in the application layer. Figure 6

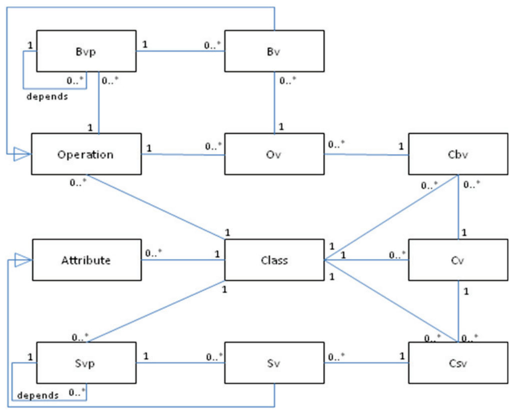


Fig. 6. Application layer meta model extension

depicts extensions to the application layer meta model highlighted in Fig 2 as follows:

- *Svp* denotes a structural variation point wherein multiple variations can be plugged.
- *Sv* denotes a structural variation for a structural variation point.
- A structural variation is in fact an *Attribute.*
- *Csv* denotes a structural variant for a *Class*. It is a complete and consistent configuration of structural variations of the class i.e. no structural variation point is left unplugged and selected structural variations honour structural variation point dependencies. Structural variants for a class differ in terms of the number of *Attributes* or their *Types* or both.
- *Bvp* denotes a behavioural variation point for an Operation wherein multiple variations can be plugged.
- *Bv* denotes a behavioural variation for a behavioural variation point.
- A behavioural variation is in fact an *Operation*.
- *Ov* denotes an Operation variant. It is a complete and consistent configuration of behavioural variants i.e. no behavioural variation point is left unplugged, and selected behavioural variants honour behaviour variation point dependencies if any.
- *Cbv* denotes a behavioural variant for a *Class*. It is a consistent configuration of Operation variants i.e., the selected operation variants serve meaningful intent.
- *Cv* denotes a variant for a *Class*. It is a type-compatible configuration of structural and behavioural variants of the class.

Thus, the above extensions enable modeling of a family of classes wherein each member (of the class family) serves the same intent in a specific situation. By making the above information available as metadata, implementation can switch from one consistent configuration of variants to another at application run-time. Not all such situational adaptations can be handled at application-runtime though, for instance, addition of a new behavior extension (*Bv or Ov*) would need recompilation (followed by redeployment). Similarly, definition of a new class altogether, as an extension to existing functionality, cannot be handled at application run-time. But, the meta model enables a new situation to be addressed such that it adds to the existing set of configuration alternatives.

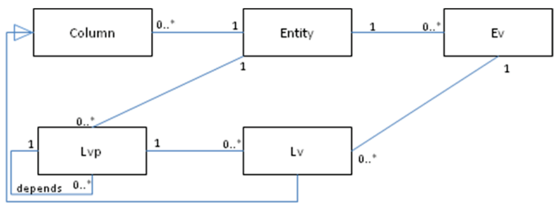


Fig. 7. Database layer meta model extension

4.1.2 Database layer

Database layer provides persistency to the desired application objects. We use object-relational database layer that provides an object façade to relational database tables implementing persistence. As an application object can vary *structurally*, the database table onto which it maps also needs to cater to this variance. And the same holds for structural extension as well. Configurability in database layer means quick switching from one known situation (i.e. db schema) to another, and extensibility means easy handling of as yet unseen situation. Figure 7 shows extension of database layer metamodel highlighted in fig. 2 as follows:

- *Lvp* denotes a structural variation point wherein multiple variations can be plugged.
- *Lv* denotes a structural variation for a structural variation point.
- A structural variation is in fact a *Column*.
- Ev denotes a structural variant for an *Entity*. It is a complete and consistent configuration of structural variations of the Entity i.e. no structural variation point is left unplugged and selected structural variations honour structural variation point dependencies. Structural variants for an Entity differ in terms of the number of *Columns* or their *Types* or both.

Thus, the above meta model enables modeling of a family of entities wherein each member (of the entity family) serves the same intent in a specific situation. In essence, the above information constitutes a generic db schema that can be specialized for a variety of situations. Database access methods such as primary-key based Create, Update, Get and Delete, complex data accesses like joins can encode interpretation of this information in their implementation. By making the above information available at application runtime, as metadata, implementation can switch from one known configuration to another at application run-time. Addition of a new row in the metadata tables corresponds to the ability of handling as yet unseen situation. Not all situational adaptations can be handled at application-runtime though, for instance, deletion of a column would need redefinition of the db schema leading to recompilation of database access layer code followed by redeployment. But, the meta model enables a new situation to be addressed such that it adds to the existing set of configuration alternatives.

4.1.3 User Interface layer

A GUI screen family represents a set of GUI screens that have a lot in common but differ from each other in a well-defined

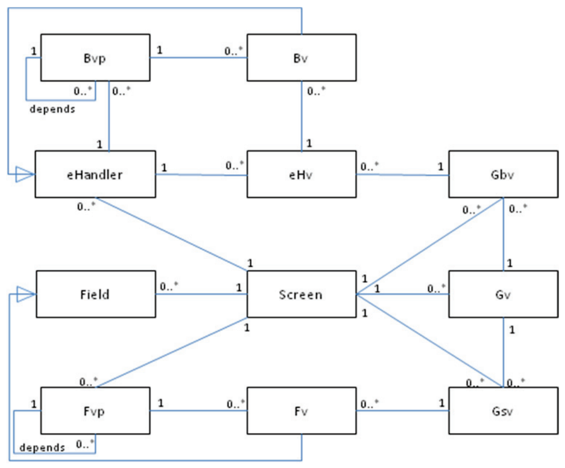


Fig. 8. User interface layer meta model extension

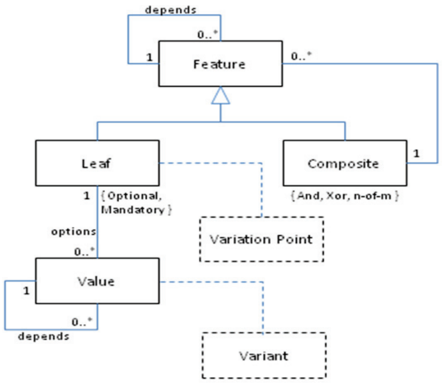


Fig. 9. Variability resolution meta model

manner. Therefore, understanding of commonality and variability is critical in order to support modeling of a GUI screen family. A GUI screen is one of the channels for users to interact with an application. In essence, a GUI screen enables user to provide input data for carrying out a logical unit of work and display the response. While serving the same business intent, a GUI screen can vary in terms of *what* (i.e. data to enter and/or view) and *how* (i.e. layout information and GUI controls to use) leading to multiple situations. Configurability means quick switching from one known situation to another, and extensibility means easy handling of as yet unseen situation. Figure 8 shows an extension of user interface meta model highlighted in fig. 2 as follows:

- *Fvp* denotes a structural variation point wherein multiple variations can be plugged.
- *Fv* denotes a structural variation for a structural variation point.
- A structural variation is in fact a *Field.*
- *Gsv* denotes a structural variant for a *Screen*. It is a complete and consistent configuration of structural variations of the screen i.e. no structural variation point is left unplugged, and selected structural variations honour structural variation point dependencies. Structural variants for a screen differ in terms of the number of *Fields* or their lay-out or both.
- *Bvp* denotes a behavioural variation point for an event handler wherein multiple variations can be plugged.
- *Bv* denotes a behavioural variation for a behavioural variation point.
- A behavioural variation is in fact an *Event handler*.
- *eHv* denotes an event handler variant. It is a complete and consistent configuration of behavioural variants i.e. no behavioural variation point is left unplugged, and selected behavioural variants honour behaviour variation point dependencies if any.
- *Gbv* denotes a behavioural variant for a *Screen*. It is a consistent configuration of event handler variants i.e. the selected event handler variants serve meaningful intent.
- *Gv* denotes a variant for a *Screen*. It is a type-compatible configuration of structural and behavioural variants of the screen.

Thus, the above meta model enables modeling of a family of screens wherein each member (of the screen family) serves the same intent in a specific situation. By making the above information available as metadata a GUI implementation can switch from one known situation to another at application run-time. Addition of a new row in the metadata tables corresponds to the ability of handling as yet unseen situation. Not all situational adaptations can be handled at application-runtime though, for instance, change in event handler code would need recompilation (followed by redeployment). Similarly, definition of a new screen altogether, as an extension to existing functionality, cannot be handled at application run-time. But, the meta model enables a new situation to be addressed such that it adds to the existing set of configuration alternatives.

4.1.4 Putting the layers together

Meta models described so far, in essence, help model a *family* at each architectural layer such that all members of a family share a *common* part and are distinguishable in terms of member-specific part. As described in section 2, the unified meta model enables specification of well-formedness constraints spanning across the architectural layers. Once a set of desired members, one from each architectural layer, is identified, it is possible to compose them into a well-formed specification that can be automatically transformed into a consistent and complete solution. We use feature modeling technique to enable selection of a family member pertaining to the desired criterion. Figure 9 shows the variability resolution meta model that captures traceability of a feature to its implementation artefacts. Augmented with constraints, this meta model enables selection of a configuration as a set of variants that is complete, consistent and well-formed. A configuration is:

– *complete* if there is no variation point remaining unplugged
– *well-formed* if structural constraints between variation point and variants being plugged therein such as Xor, And, n-of-m are satisfied
– *consistent* if variants honour dependency constraints between their variation points

In our experience this simplistic variability resolution meta model has sufficed so far. We are aware it will need to be richer going forward.

4.2 Experience and lessons learnt

We are in the early roll-out stage of this solution. In the interest of time-to-market, product owners decided to use the latest product release as baseline for introducing the family concept as opposed to refactoring the set of solutions delivered so far into a productline. Since new meta models are essentially an extension of old meta models, it was possible to migrate the older application models fully automatically. Given the simple nature of meta model extensions, full power of QVT [13] was not called for and a simpler imperative model transformation alternative [14] sufficed. In early experience, modeling of commonality and variability rooted at meta objects being used for code generation, namely, *Class*, *Entity*, *Operation* and *Screen* seem to suffice. Proposed meta models specify pre-defined variation points, possible variations, and constraints over variation points. Configuration is a process of selecting appropriate variations so that all variation points are consistently plugged-in for an application. We supported this configuration process at three different stages of application development: design time, installation time and run time. Design time configuration is supported through model transformation and model merge techniques. The installation time and run time configuration is supported by generating appropriate metadata for all possible variations. Having separated business process concern from application functionality, we had to support the family concept for business process models as well [19]. We think the true test of the proposed meta models as regards configurability and extensibility will come in supporting inherently dynamic domain of financial instruments, insurance products etc.

5. RELATED WORK

The idea of addressing a set of related situations in an integrated manner is not new. Parnas was the first to argue for the need to design software for ease of extension and contraction thus leading for software families [3]. Usual practice is to parameterize the solution so as to address known situations. Several approaches for supporting parameterization through variability management have been proposed. Extending UML for modeling variability using standardized extension mechanisms of UML is presented in [11]. A variation point model that allows extension of components at pre-specified variation points is proposed in [6]. A conceptual model for capturing variability in a software product line is presented in [4]. All the three only support the notion of variation point and that too only at modeling level whereas we provide support for structural and behavioural levels. Aspect-orientation [5] is a technique for addressing separation of concerns with greater modularization and locality. However, implicit communication link between aspects and classes complicates the readability and comprehension of an aspect-based realization of variability architecture [1]. We circumvent this problem by generating pure OO implementation with aspects suitably woven in. Feature modeling is a popular mechanism to specify product lines [9]. Although a feature model can represent commonality and variability in a concise taxonomic form, features in a feature model are merely symbols. Mapping features to other models, such as behavioral or data specifications, provides a path towards implementation. A general template-based approach for mapping feature models to concise representations of variability in different kinds of other models is presented in [7]. We build upon this idea to connect a feature to its implementation artefacts i.e. structural and behavioural specification through a meta model. Despite years of progress, contemporary tools often provide limited support for feature constraints and offer little or no support for debugging feature models. An integration of prior results to connect feature models, grammars, and propositional formulas so as to allow arbitrary propositional constraints to be defined among features and enable off-the-shelf satisfiability solvers to debug feature models is presented in [2]. We build upon these ideas to ensure consistency of the selected feature configuration. The meta model connecting features with their implementation artefacts guarantees consistency, correctness and completeness of the implementation.

6. SUMMARY

We presented a model-driven generative approach to address adaptation needs of business applications. The approach builds further on the ideas of separation of concerns, variability management, feature modeling and generative development. We visualize business applications to vary along five dimensions, namely, Functionality (F), Business process (P), Design decisions (D), Architecture (A) and Technology platform (T). We address adaptation needs along all the five dimensions using a specification-driven generative approach. We extend meta models to support specification of variability along F and P dimensions. We presented a meta model that connects these specifications to features in a feature model. This bridge meta model enables traceability of a consistent well-formed feature configuration to its specification artefacts thus realizing a family of application specifications corresponding to the feature model. We presented building block abstraction as a mean to implement a D or A or T feature. MDD code generator is a hierarchical composition of building blocks, and maps easily to a consistent well-formed configuration of a feature model along D, A and T dimensions.

We discussed our experience in using model-driven techniques to build large business applications on a variety of architectures and technology platforms. Separating business functionality from technological concerns, and model-based code generation resulted in significant productivity and quality gains. Encouraged by these benefits, many large development projects also readily adopted the model-driven approach despite initial investment in learning how to model. This enthusiastic, and somewhat unexpected, acceptance of the approach led to an ironical situation of the productivity toolset team becoming a bottleneck. We overcame this problem through use of product line techniques in order to model the code generators as a family, and deriving a purpose-specific implementation therefrom. Thus, we could achieve scale through addressing customizability at family level instead of individual member level.

We discovered the same issue with business functionality i.e. solutions delivered to different players in the same business domain were not exactly alike even while addressing the same business intent. We shared early stage experience of modeling commonality and variability along F and P dimensions which seems encouraging.

Though the idea of bringing together separation of concerns, variability management, and feature modeling seems promising, there are several open issues:

– The meta model providing traceability from F and P features to their implementation specifications is rather simplistic.

– There should be support, preferably tool-aided, for unit testing a feature - it should be possible to specify test cases for a feature independently and compose the test cases to arrive at the system level test cases for the desired feature configuration.
– There should be tool support for intelligent debugging at feature level. A bug detected at code level should be traceable back to the feature specification.
– Hierarchical organization of features enforces an ordered traversal. Complex interdependence of features may impede strict order.
– It is not clear which facets of a system deserve to be modeled as building blocks. There is a need to investigate how the engineering aspects can be modeled and what the right kind of abstractions for modeling them are to satisfy the various 'ities' like maintainability, reusability etc. For instance, how does one model a design for better maintainability?
– Building blocks may overlap each other thus introducing an order of weaving. How does one ensure that properties of all building blocks hold after their weaving?
– Supporting separation of concerns using building blocks raises several tooling issues. The modeling tool should be extensible to support new modeling languages so as to define new aspect models and relate them to existing models. The model transformation tool should have adequate support for pattern matching and composition. The tool should support incremental reconciliation of models and scale up to cater to the demands of enterprise class applications.

In comparison to the existing literature, the proposed approach centered around meta models capturing commonality and variability in all dimensions of a typical database intensive business application seems more pragmatic for industry use. We are working on development of a component abstraction and algebra to support configuration and extension operators for these dimensions. Also, going forward we hope to ride piggy-back the technology advance in OSGi [15].

7. REFERENCES

[1] Alexander Nyßen, Shmuel Tyszberowicz, Thomas Weiler. Are Aspects useful for Managing Variability in Software Product Lines? A Case Study. Early aspects workshop at SPLC 2005.

[2] Don Batory. Feature Models, Grammars, and Propositional Formulas. Software Productlines, Volume 3714 of LNCS, pages 7-20, Springer, 2005.

[3] D L Parnas. Designing Software for Ease of Extension and Contraction. Proceedings of the 3rd ICSE, pages 264 – 277,1978.

[4] Felix Bachmann, Michael Goedicke, Julio Leite, Robert Nord, Klaus Pohl, Balasubramaniam Ramesh and Alexander Vilbig. A Meta-model for Representing Variability in Product Family Development. Software Product Family Engineering, volume 3014 of LNCS, pages 66-80, Springer, 2004.

[5] Gregor Kiczales, John Lamping, Anurag Mendhekar, Chris Maeda, Cristina Videira Lopes, Jean-Marc Longtier and John Irwin. Aspect oriented programming. ECOOP'97 LNCS 1241, pp 220-242. Springer-Verlag. June 1997.

[6] Hasan Gomaa, Diana L Webber. Modeling Adaptive and Evolvable Software Product Lines Using the Variation Point Model. Proceedings of the 37th Annual Hawaii International Conference on System Sciences (HICSS'04) - Track 9 - Volume 9.Page: 90268.3

[7] K. Czarnecki and M. Antkiewicz. Mapping features to models: A template approach based on superimposed variants. Generative Programming and Component Engineering, Volume 3676 of LNCS, pages 422–437. Springer, 2005.

[8] K Czarnecki and U Eisenecker, Generative programming methods, tools and applications, Addison-Wesley, 2000.

[9] K Kang, S Kohen, J Hess, W Novak and A Peterson, Feature-orientation domain analysis feasibility study, Technical Report, CMU/SEI-90TR-21, November 1990.

[10] MasterCraft – Component-based Development Environment. Technical Documents. Tata Research Development and Design Centre. http://www.tata-mastercraft.com

[11] M Clauß, I Jena. Modeling variability with UML. GCSE 2001Young Researchers Workshop, 2001.

[12] MOF Models to Text Transformation Language http://www.omg.org/spec/MOFM2T/1.0/

[13] MOF Query / Views / Transformations http://www.omg.org/spec/QVT/1.0

[14] OMGen Reference manual, version 1.5, Technical Document, Tata Consultancy Services, May, 2008

[15] OSGi - The Dynamic Module System for Java, http://www.osgi.org/

[16] UML Infrastructure 2.0 Draft Adopted Specification, 2003, http://www.omg.org/spec/UML/2.0/

[17] Souvik Barat and Vinay Kulkarni: Developing configurable extensible code generators for model-driven approach. 22nd International Conference on Software Engineering and Knowledge Engineering, July, 2010.

[18] Vinay Kulkarni, R. Venkatesh and Sreedhar Reddy. Generating enterprise applications from models. OOIS'02, LNCS 2426, pp 270-279. 2002.

[19] Vinay Kulkarni and Souvik Barat: Business Process Families using Model-driven Techniques. 1st International workshop on Reuse in Business Process Management, Sep, 2010. http://each.uspnet.usp.br/rbpm2010/program.htm

[20] Vinay Kulkarni, Sreedhar Reddy, An abstraction for reusable MDD components: model-based generation of model-based code generators. GPCE 2008: 181-1843.

[21] Vinay Kulkarni, Sreedhar Reddy: Introducing MDA in a large IT consultancy organization. APSEC 2006: 419-426.

[22] Vinay Kulkarni, Sreedhar Reddy: Model-Driven Development of Enterprise Applications. UML Satellite Activities 2004: 118-128

Dynamically Adaptable Software Product Lines Using Ruby Metaprogramming

Sebastian Günther and Sagar Sunkle
School of Computer Science
University of Magdeburg, Germany
sebastian.guenther@ovgu.de
sagar.sunkle@ovgu.de

ABSTRACT

Software product lines (SPL) is a paradigm to structure software development assets in a common and reusable form. Out of this common asset base – which includes the application's source code, documentation, and configuration – concrete product variants can be created. The variants are differing in terms of the features, which are basically an increment in functionality important for a stakeholder. Feature-oriented programming (FOP) provides the capability to compose those different variants. In earlier work we presented rbFeatures, a FOP implementation in Ruby. With rbFeatures, features become are first-class entities of the language that facilitate runtime changes of the program. This paper presents an extension to rbFeatures that implements product lines and their variants as first-class entities too. The entities allow powerful runtime-adaptation and configuration, like to add new features or constraints to the product line and the instantiation of several variants with different feature configurations. The particular contributions are to show how Ruby's metaprogramming capabilities are used to design first-class entities and an explanation of the usage of our approach with a common case study.

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, Software Product Lines, Metaprogramming, Domain-Specific Languages, Runtime Adaptation

1. INTRODUCTION

In software engineering, features provide additional modularization to applications. Features are special characteristics of software which distinguish members of a program family [2]. Program families evolved to today's understanding of software product lines that share a "common, managed set of features" [26]. In order to provide a more structured approach to software design and implementation, features and other valuable production assets are grouped together to form software product lines [3]. Recently, software product lines with dynamic adaptation facilities are gaining a widespread interest [27, 21, 18]. The adaptability need includes the reconfiguration of variants at runtime and the instantiation of new variants. On motivation for such runtime adaptation is the seamless migration of a 24/7 application when the codebase was updated with new feature code [21].

We see features from a conceptual and an implementation viewpoint. Conceptually, *model features* help to structure the production assets in the form of a product line. At the implementation level, a *concrete feature* is the sum of all code inside a program that belongs to a particular model feature. An important consideration in our research is the idea to provide first-class entities. When features become entities of the program itself, they help to bridge the gap between the conceptual and the implementation level.

In earlier work [8], we presented a FOP implementation in Ruby called rbFeatures. With rbFeatures, developers add feature containments around selected blocks of code. The feature containment condition specify under which feature configuration these parts are active. Only if the condition is satisfied, the contained code will be contained in the application. The functionality provided by rbFeatures includes runtime re-configuration of the application by activating or deactivating features and runtime extension of the program and its features.

Since then we extended rbFeatures. The current version extends rbFeatures with a language for product line modeling. The language's intent is to represent the known tree-like feature models that express an application's model features and their relationships [3], as well as their constraints like mandatory and optional features. By combining rbFeatures with this language, we implemented abstractions for features, product lines, and variants as first-class entities. This particular solution provides rich runtime adaptation and configuration of software product lines, including the provision of multiple variants and variant modification.

This paper provides a complete coverage of feature modeling, feature implementation, and dynamic runtime composition and modification of a software product line and its variants. The particular contributions are to show how Ruby's metaprogramming capabilities are used to design first-class entities and an explanation of the usage of our approach with a common case study. By describing how we utilized the host language Ruby to build the extension, we wish to show other FOP researchers how to build first-class entities and how this approach supports the goal of features and runtime adaptation.

In Section 2, further background about regarding feature-oriented programming, software product lines, and dynamically adaptable SPL's is explained. Section 3 elaborates the basics of the extended version of rbFeatures, and in Section 4 we explain how first-class variants are implemented as objects and how they can be used for runtime adaptation and customization. Section 5 gives related work and Section 6 summarizes this paper. We apply following formatting: *keywords*, FEATURES, and `source code`.

2. BACKGROUND

2.1 Feature-Oriented Programming

Features can be seen from two perspectives. The fist perspective regards features as all parts of a software that reflect the concerns of a stakeholder [3]. Features are "common aspects [...] as well as differences between related systems" [10]. Features are important in domain engineering to scope the software and they also provide the stakeholder-important requirement of an application. These features are called *model features.*

The second perspective expresses features at the code base – we call them *concrete features.* In this perspective, FOP is a paradigm that was introduced as a "new conceptual model for object and object composition" [20]. It allows grouping and composing sets of classes to obtain different variants of a program. How to implement features or compose variants out of features is an open research field. Since its inception, many FOP implementations have been proposed: Mixin-layers [23], AHEAD-refinements [2], and aspectual feature modules [1] to name a few. In general, the approaches can be divided into two different forms [12]. In the *compositional approach*, features are added as refinements to a base program. The explicit representation of all code belonging to a feature is an expression of the separation of concerns principle [4]. In the *annotative approach*, features are represented as annotations inside the source code. The representation can be implicit on top of the source code like in CIDE [11], which uses a representation on top of the programs abstract-syntax tree, or it can be explicit by using language constructs as in our rbFeatures approach [8].

While both perspectives are certainly providing benefits, the still existing gap between the two representations is to be questioned. We argue that a complete representation of model and concrete features provides the benefits of both worlds: A clean structuring of stakeholder concerns and tooling to prioritize development decisions, as well as the technical capability to build and deliver custom variants with respect to the available configurations as expressed with the software product line model. Section 3 explains how to implement this vision.

2.2 Software Product Lines

When developing software, one fundamental decision is to design either one-off systems or a program family. One-off systems are scoped, configured, and executed for one exact purpose. After its development, the system goes into a stable usage and maintaining period, and eventually is replaced by an successor. On the contrary, program families [4] are applications that are used in different configurations for similar, but not the same purposes. Members of a program family have several *commonalities* with their members while the *variable* part exhibits the different configurations.

Software product lines is the modern name for software families with a special focus on the providing automatic means to derive individual variants from a common codebase. The need for product lines stems from today's strong individualization requirements that drive customization and software flexibility to its height. As [26] explains, "managers must invest strategically in software assets to gain competitive advantage in the battlefield or the marketplace". Following this need, SPL identify, structure, and provide a set of production assets that are systematically reused [3]. The connection between software product lines and features is a compositional one: "product line is a group of products sharing a common, managed set of features" [26]. A feature-diagram can be used to represent the relationships and constraints between the features in a tree-like structure. A particular configuration of features is a valid variant if all the constraints specified in the feature model are satisfied.

Feature-oriented programming is one option to provide the product-line feature structure for the assets, especially for assets related to implementation. Dependent on the particular FOP implementation, this allows different representations of the product line, constraints, features, variants and composition approaches.

2.3 Dynamically Adaptable Software Product Lines

Software product lines with dynamic adaptation facilities are gaining a widespread interest in recent publications [27, 21, 18]. The primary motivation for having runtime adaptation is to provide different variants that support specific application needs. In one case study, complex Enterprise Resource Planning Systems are configured on-site in customer sales acquisition [27]. In the sales dialog, customers express their requirements. The presenter customizes the application accordingly, and the customers can test the application and refine their requirements until they are satisfied. Another use case of dynamic adaptation is to support 24/7 applications [21]. In order to continually evolve the application without providing any downtimes, one approach is to enable the live-update of the running application. Once a new feature has been implemented, the running application is carefully migrated to the new version. In this process, the product line model helps to maintain the structural relationships between the assets and can be used for testing prior to deployment.

3. rbFeatures

rbFeatures [8, 7] enables Feature-Oriented Programming with the Ruby programming language. Features become first-class entities of a program. They are constants that can be used in any expression and are thus open to runtime

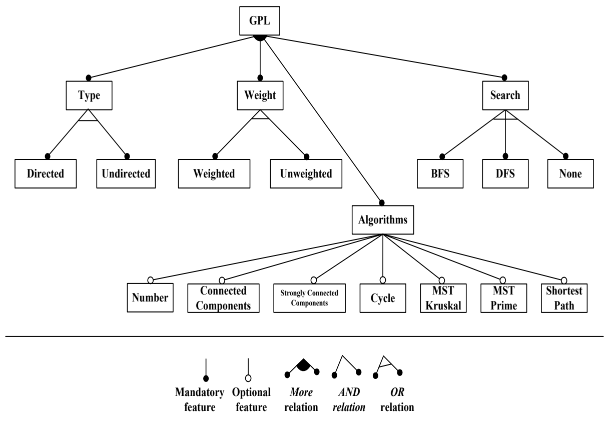


Figure 1: Feature model of the GPL.

modification too.

This section details the basic implementation and application of rbFeatures. Thereby, we use the Graph Product Line (GPL) as the ongoing example. The GPL is a product line that provides different variants for graphs and graph algorithms [15]. We see the tree-like structure of the GPL in ►Figure 1. As can be seen, the product line differentiates the type and weight of a graph, provides search algorithms, and implements numerous other algorithms like determining whether the graph is a connected graph.

3.1 Central Entities

The initial version of rbFeatures consisted of two central entities. The `Feature` module contains all methods that form the public API of features and internal functionality. This module is mixed into normal classes that represent an application's features. The second central entity is the `FeatureResolver`. It defines the required background functionality so that the code associated with a feature or a combination of features is active with regard to the feature configuration.

The extension of a domain-specific language [16] for product line modeling that we developed in an earlier paper [5] adds additional entities. A `FeatureModel` is a configuration unit that represents model features. It contains a name, a list of subfeatures, the position it has in the feature tree, and constraints. The `ProductLine` entity is defined by adding all configured model features. Finally, the `ProductVariant` entities represent a concrete feature-configuration and can be instantiated at runtime.

The relationships between all entities is shown in ►Figure 2. Using the extended version of rbFeatures encompasses the following steps:

1. *Product Line Modeling* – Define the model features, their relationships, and their constraints. Add all model feature to a `ProductLine` object.
2. *Application Implementation* – Implement or feature-refactor an application.
   - Create `Feature` objects that represent the identified the model feature.
   - Form feature containments by enclosing all code parts in a block and provide a containment condition expressing under which feature configuration the containment is executed.
3. *Program Initialization* – Initially execute the program to obtain the first-class representations for the product line, model features, and the concrete features.
4. *Variant Creation and Instantiation* – Create *ProductVariant* objects by configuring a set of activated features that are valid to the feature model, and then instantiate the variant in a defined scope.

The next subsections explain each step in detail.

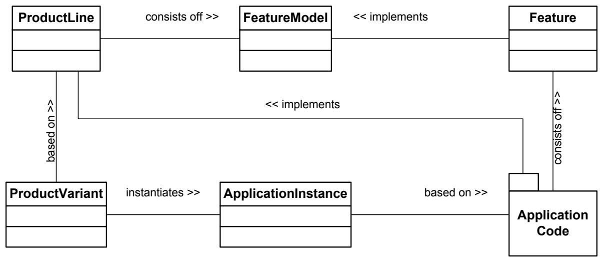


Figure 2: Entity structure of rbFeatures.

3.2 Product Line Modeling

The first step is to provide the model features and the productline. Each model feature needs to be declared with the following properties: `name`, position in the tree (`root`, `node`, `leaf`), the `subfeatures`, and a set of constraints (using the keywords `is`, `one`, `more`, `all`, `any`). For example, the declaration of the root node `GPL` takes the following form in ►Figure 3.

This declaration is easy to read for itself. Beginning in Line 2, the name of the feature is defined, its relative position in the feature tree declared, its subfeatures listed, and finally a constraint defined. The constraint expresses that the selection of the GPL feature requires selecting one ore more of it's subfeatures (`Type`, `Weight`, `Search`, and `Algorithms`).

After all features are declared in this way, we can declare the product line object. It uses the syntax shown in ►Figure 4. After a short description in Line 2, the next

```
1  gpl_feature = FeatureModel.configure do
2    name :GPL
3    root
4    subfeatures :Type, :Weight, :Search, :Algorithms
5    requires :GPL => "more :Type, :Weight, :Search,
         :Algorithms"
6  end
```

Figure 3: Configuring the root feature of the GPL.

```
1 GPL = ProductLine.configure do
2   description "The complete GPL"
3   add_feature gpl_feature
4   add_feature type_feature
5   add_feature weight_feature
6   #...
7 end
```

Figure 4: Creating a product line by adding all features.

```
class Weighted
  is Feature
end

class Unweighted
  is Feature
end

class Undirected
  is Feature
end
```

Figure 5: Implementing basic features.

lines just add the model features to this product line. Once the product line declaration is complete, the `valid?` method checks whether all named subfeatures and features contained in the conditions are included – this helps to detect incorrect product line objects.

After these first steps we have a complete set of model features and the product line available. The next step is to provide the implementation of the product line, and later combine the model and the implementation for creating variants.

3.3 Application Implementation

The next step is to declare the concrete features. Either a complete application including features is developed from scratch, or an existing application is feature-refactored. In both cases, the task is to first define the features entities and then to form feature containments.

As we explained before, concrete features are normal classes. If they do not contain functionality on their own, their declaration is as simple as shown in ►Figure 5. If they contain functionality on their own, then the class declaration body includes additional fields and methods as it is commonly defined in Ruby.

Once the features are defined, we use them to form feature containments. Containments consist of two parts. The first part is called *containment condition*. It is an expression that specifies which activation status one or several features require. Conditions like "If feature A, B and C, but not D are activated", translate to the natural syntax `(A + B + C - D)`. The second part of the feature containment is the *containment body*. It contains code that belongs to the particular feature or the intersection of features specified in the condition. rbFeatures supports three granularity levels of feature code: (i) complete class or module declarations, (ii) method declarations, and (iii) individual lines or even individual characters in lines. The granularity of the containment condition and the containment body enacts high flexibility of declaring code for a single feature or interacting features.

The GPL requires to put both lines containments and complete method bodies in feature containments. As shown in ►Figure 6, the `Weight` feature interacts with `Edge` by defining the local `weight` variable and an accessor for it (Line 2 and 4). The features `Directed` and `Undirected` interact with `Graph` by defining the `directed?` method with a custom body (Line 11–16 and 17–21).

Once the application is completed with all features and feature containments are created, we can initialize the program.

```
 1 class Edge
 2   Weighted.code { attr_accessor :weight }
 3   def initialize(params)
 4     Weighted.code { @weight = params.delete :weight }
 5     params.delete :weight if params.include? :weight
 6     #...
 7   end
 8 end
 9
10 class Graph
11   def initialize(gtype)
12   Directed.code do
13     def directed
14       return gtype == 1
15     end
16   end
17   Undirected.code do
18     def directed
19       return false
20     end
21   end
22   #...
23 end
```

Figure 6: Implementing the GPL application with feature containments.

3.4 Program Initialization

For providing runtime changes to the application, rbFeatures requires a complete representation of the application. Because of the mechanisms used to instantiate a variant in a specific namespace, we use string representation. A `ProcductLine` object stores this string representation and can invoke it several times to define new variants.

Once the code's representation has been provided, the next step is to initially execute it and to define the applications modules, classes, objects, and the features. In this process, we use several hooks and metaprogramming mechanisms that help to change specific parts of Ruby's normal behavior. The most important ones are listed here:

- *Initial execution of feature containments* – Initially all features are deactivated, so that the normal initialization of the program would not execute feature containments. However, whole modules and classes would not be available, and several methods not defined. This could lead to an in-executable program. Because of this, all containment bodies are executed nevertheless, and thus the program is initialized with all entities and methods defined. However, the methods are actually not executable: The method-added hook modifies them.
- *Method-added hook* – Ruby provides several hooks that are called at specific runtime conditions (a complete list is explained in [25]). rbFeatures uses a hook that is triggered whenever a new method is added to an object. The hook checks if there was a feature violation stemming from the last containment condition. If yes, it replaces the method's body with a custom error message specifying the conflicting feature (for example "FeatureNotActivatedError: Feature DFS is not activated").
- *Instantiation prohibition* – Deactivated features are not allowed to create instances. In the initialization phase, we overwrite the `initialize` method to throw an error too, using the explained method-added hook.

3.5 Feature Activation and Deactivation

After initialization, the program consisting of all classes, modules, and methods exist. Yet the provided functionality is limited because all features are deactivated by default. Activating or deactivating features changes the program[1]. Each time a feature changes its activation status, the `FeatureResolver` is triggered to re-evaluate[2] the string representation of the application code. This means to execute all module, class, and method declarations again. Eventually, the containment conditions are now valid, and methods previously not available can now be executed normally.

This modification uses two important metaprogramming capabilities of Ruby: *open classes* and *code evaluation.*

- *Open classes* – Ruby allows modifying all existing entities, even the built-in ones. For example if a method declaration is executed in the same scope and with the same name as an existing method, then the old method is overwritten.
- *Code evaluation* – At runtime, code in the form of `String` or `Proc` objects can be evaluated. Strings are an external format and are slower to evaluate, but using Ruby's string processing capabilities they can also be changed arbitrarily. Procs instead are transformed to an internal representation. Like strings, they can contain any expressions, but are not modifiable after their creation[3] The difference is that procs are similar to closures stemming from functional-oriented programming. Their declaration encloses the state of surrounding variables even if the original context is no longer available.

Both concepts explain the dynamic adaptation capabilities of rbFeatures. At first, the whole application is stored inside a string object, giving full manipulation capabilities of the code with built-in string processing capabilities. Second, containment bodies are actually proc objects, defined with the `do` ... `end` notation shown in previous examples. And third, every time a feature changes its activation status, the complete application is re-evaluated again. Parts of the application that were not available before may get active, and this changes the internal program representation.

3.6 Variant Creation and Instantiation

We consider the case to create a `ProductVariant` object that provides the features DIRECTED, WEIGHTED, DFS, and STRONGLY CONNECTED. The expression to create this object are shown in ►Figure 7. The variant receives a name, a parameter pointing to the `ProductLine` model, and in its body various features are activated. When the variant is created, it is automatically added to the product line and can be retrieved from there.

[1]In the current version of rbFeatures, manual configuration using the first-class feature objects is still possible. But this should be used with care because no product line model is available and therefore the feature configuration is not checked. This could lead to buggy programs. Therefore, the use of `ProductVariant` objects as explained in the next subsection 3.6 is recommend.

[2]The method `eval` is used to execute `String` or `Proc` representations of Code, hence we speak of evaluation.

[3]At least with the standard library. In [7] we showed how an external library can be used to obtain a string representation of a a proc, to modify it, and to writ it back to the proc.

```
ProductVariant.configure
  :name => "SimpleVariant",
  :pl => GPL do
    activate_features :Directed,
                      :Weighted,
                      :DFS,
                      :Strongly_Connected
  end
end
```

Figure 7: Configuration of a product variant.

Once the variant is available, a call to it's `instantiate!` method actually creates an instance of this variant. This triggers the following steps:

1. Check whether the configured features are valid to the product line model by checking that all specified constraints are satisfied.
2. Compose a string template consisting of a module and the product line code. The module uses the configured name of the variant and serves as a namespace.
3. Add the string template to the core entity `FeatureResolver` (see Section 3.1).
4. Initialize the application by evaluating the template once.

From this moment on, the code contained in the variant is available at its separate scope, and all feature activation changes are governed by the `FeatureResolver` and the `ProductLine` model. Whenever a feature's activation status is changed and possibly a variant's instance modified, the new configuration is checked with the product line model. Only valid feature configurations are allowed. When variants are used, rbFeatures synchronizes the various methods that change a feature's activation status with each other. For example, changing the feature status in the variant or adding a new feature is immediately reflected in the instance. Also, changes of features directly in the variant instance also synchronize with the variant object.

Now we will see the application of these changes in an example.

4. RUNTIME ADAPTATION EXAMPLE

In this section we give an example on how to use the facilities of rbFeatures for providing multiple variants and runtime adaptation.

We assume the GPL example is completely implemented in terms of the product line model, the application code, and concrete features. Then two variants are created using the `ProductVariant` entity in ►Figure 8, Line 1–15. The `ShortestPathVariant` includes the features WEIGHTED, DIRECTED, and SHORTEST PATH, and the *DFSVariant* Includes the features WEIGHTED, DIRECTED, and DFS.

The next step is to instantiate the variants and to create graphs inside them. To create the instance and a graph object for the `ShortestPathVariant`, the expressions in ►Figure 8 (Line 17–29) are used. First, the variant needs to be instantiated with the `instantiate!` method. Second, we select the variant by using the `variant` method of the product line object. Third, a code block is executed in Line 10–28 to create the graph. The `DFSVariant` is created similarly in ►Figure 8 (Line 33–42), but it receives a simpler graph. The resulting

```
ProductVariant.configure
  :name => "ShortestPathVariant",
  :pl => GPL do
    activate_features :Weighted,
                      :Directed,
                      :ShortestPath
  end
end

ProductVariant.configure
  :name => "DFSVariant",
  :pl => GPL do
    activate_features :Weighted,
                      :Directed,
                      :DFS
end

GPL.variant("ShortestPathVariant").instantiate!

ShortestPathVariant.class_eval do
  graph = Graph.new
  1.upto(6) { |n| graph + node(n) }
  graph + edge(1 => 2, :weight => 1)
  graph + edge(1 => 3, :weight => 2)
  graph + edge(2 => 5, :weight => 4)
  graph + edge(2 => 4, :weight => 2)
  graph + edge(4 => 6, :weight => 12)
  graph + edge(3 => 6, :weight => 22)
  SPgraph = graph
end

GPL.variant("DFSVariant").instantiate!

DFSVariant.class_eval do
  graph=Graph.new
  1.upto(6) { |n| graph + node(n) }
  graph + edge(1 => 2, :weight => 1)
  graph + edge(2 => 3, :weight => 2)
  graph + edge(2 => 4, :weight => 3)
  graph + edge(3 => 5, :weight => 4)
  graph + edge(4 => 5, :weight => 4)
  DFSgraph = graph
end
```

Figure 8: Creating two distinct variant objects and instantiating them.

graphs are shown ►Figure 9 – `ShotestPathVariants` to the left, and `DFSVariant` to the right.

Now that both variants are created and instantiated, we can use and modify them at will. In ►Figure 10, following modifications are applied.

- Test whether each variant has an independent graph object (Line 1 and 2).
- In the `ShortestPathVariant`, calculate the shortest path between the node 1 and 6 and show the result (Line 5).
- Also, try to calculate the shortest path between node 1 and 5 in the `DFSVariant` (Line 9). But this raises an error, demanding that the ShortestPath feature needs to be activated in this variant first.
- Activate the ShortestPath feature (Line 12).
- Now the shortest path can be calculated and the result shown (Line 15).

There is no limit to the number of product lines, variants, and variant instances that can be created (except physical borders like available memory). Also, the first-class objects can be changed at will, including runtime updates of the product line model, which are reflected back down to the instances. Here are some more examples how to use the

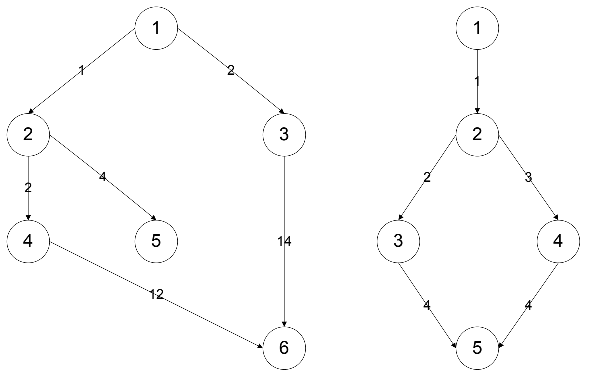


Figure 9: Resulting graph structures.

```
1  ShortestPathVariant.eval { SPgraph.length } # => 6
2  DFSVariant.eval { DFSgraph.length } # => 5
3
4  ShortestPathVariant.class_eval do
5    shortest_path(SPgraph, 6) # => [1,2,4,6]
6  end
7
8  DFSVariant.class_eval do
9    shortest_path(DFSgraph, 5) # =>
         FeatureNotActivated Error: Feature
         ShortestPath is not activated
10 end
11
12 GPL.variant("DFSVariant").activate_features
       :ShortestPath
13
14 DFSVariant.class_eval do
15   shortest_path(DFSgraph, 5) # => [1,2,3,5]
16 end
```

Figure 10: Runtime adaptation of `ProductVariant` objects.

flexibility given by the first-class entities:

- Define new model features and add them to a product line.
- Inside a variant, add a new block of code that contains a new concrete feature.
- Offer different variants to the user and record which feature combinations are heavily used. Use this information to prioritize development of additional features.
- Analyze how a particular variant is used, change its implementation at runtime by activating a new feature and changing the instance.

5. RELATED WORK

In the research field of dynamic runtime adaptation, several examples and approaches have been presented. The following subsections explain general frameworks, implementation mechanism, and applications.

General Frameworks

One general framework for runtime adaptation expresses three important concerns: (*1*) explicit architectural model, (*2*) provision of structural and behavior constraints, and (*3*) the availability of software connectors for runtime changes

[18]. In rbFeatures, concerns 2 and 3 are expressed with the help of the software product line constraints and the explained concepts of open classes and runtime code evaluation. Furthermore, concern 1 is of no importance in rbFeatures because the feature containments are applicable to any source code independent of its place in the software architecture.

Another work suggests an application-independent and generic meta-model for runtime adaptation [17]. The model considers systems operations, services, and ports of an application as possible entry points for changes, and explains that a concrete binding and implementation for each one can support the adaptation. This generic model can explain how several concrete adaptation approaches work. Recently, rbFeatures was applied to web applications where different feature configuration determine the available functions and web-pages [6]. Altering the behavior of HTTP request handlers and the offered ports is similar to providing a service, and thus rbFeatures can be seen as lightweight instance of this approach too.

Finally one model actively suggests to use features as the dominant entities that drive runtime adaptation [14]. The paper introduces a feature model, binding units, and a feature binding graph which is used to backup the stepwise change of the current features configuration. This is supported by rbFeatures too, albeit the changes in terms of checking the feature configuration with the product line model are comparatively simple.

Implementation Mechanisms

Looking at concrete implementation mechanisms, [22] explains an approach how to enable features for static and dynamic binding. Static binding is a a-priori deployment decision that confines some parts of the application to one specific configuration. Dynamic binding flexibilizes the development of features on the one hand, but it introduces additional overhead like memory consumption or runtime performance degradation. The paper shows how to use the delegator pattern and refinements using binding units can combine both approaches. This is an interesting idea to provide other mechanisms in Ruby as well. Although there is not compile time in Ruby, C extensions to the interpreter could be written that allow the pre-configuration of an application.

Another approach uses Java in order to adapt a software product line and its variants at runtime [21]. Two mechanisms to add new code based on a changed feature model. At first, existing classes are replaced with a similar, slightly evolved class but with another name through the classloader. Second, Java HotSwap is applied to change all method calls that have a callee which is the modified class. How this specific technique can be used for Feature-Oriented Programming in Java is shown in [24]. Ruby supports such runtime modification and metaprogramming mechanisms out of the box, so rbFeatures just needs to use the existing mechanisms.

Applications

Finally considering concrete applications, we see that [27] explains a plug-in based adaptation mechanism on top of the .NET platform. In this paper, some scenarios where runtime adaptation is required or beneficial are explained. One example is a live sales presentation of an enterprise resource planning system. At the presentation, the system is dynamically configured according to obtain the best feature combination which satisfied most or all requirements of the customer. Another example explains a product line which is customized in accordance with the physical and execution environment [19]. The paper presents a case study in which an application displays information about movies. The application features a cache which is activated once the bandwidth of an internet-connection reaches a certain threshold. From thereon, the application serves the data out of the cache instead of live from the server.

6. SUMMARY

This paper explained how feature-oriented programming and runtime adaptation of variants can be achieved by using first-class entities. The approach uses Ruby as the implementation language. By using existing objects (classes and modules), metaprogramming capabilities (open classes, runtime code evaluation, hooks) and functional programming (support of closures as anonymous code blocks), powerful first-class representations of product lines, features, and variants can be created. This approach is generalizable to bring runtime adaptation support for product line variants to other applications as well. Provided the chosen host languages supports similar mechanisms, first-class entities can be created in other languages as well.

A future research direction is to climb the ladder of available abstractions even higher. Once features, product lines, and variants become first-class entities of a host language, they build a reflexive layer about applications. The applications are becoming an abstraction which enables fine-grained modifications and runtime adaptation. By extending the reflexive layer with additional concerns and paradigms, such as aspect-oriented programming [13] or context-oriented programming [9], whole systems comprising several applications can be expressed with powerful meta-expressions. This idea can be easily visualized as building a product line that consists of other product lines. A system as a whole is adaptable by modifying its components which are variants of individual product lines. This architecture can be used to react to complex environmental changes.

Acknowledgments

We thank the anonymous reviewers for helpful comments on an earlier draft of this paper.

7. REFERENCES

[1] S. Apel. *The Role of Features and Aspects in Software Development.* Dissertation, Otto-von-Guericke-Universität Magdeburg, 2007.

[2] D. Batory, J. N. Sarvela, and A. Rauschmayer. Scaling Step-Wise Refinement. In *Proceedings of the 25th International Conference on Software Engineering (ICSE)*, pages 187–197. IEEE Computer Society, 2003.

[3] K. Czarnecki and U. W. Eisenecker. *Generative Programming: Methods, Tools, and Applications.* Addison-Wesley, Boston, San Franciso et al., 2000.

[4] E. W. Dijkstra. *Notes on Structured Programming.* Academic Press Ltd., London, 1972.

[5] S. Günther. Engineering Domain-Specific Languages with Ruby. In H.-K. Arndt and H. Krcmar, editors, *3. Workshop des Centers for Very Large Business*

Applications (CVLBA), pages 11–21, Aachen, 2009. Shaker.

[6] S. Günther. Multi-DSL Applications with Ruby. *IEEE Software*, 27:25–30, 2010.

[7] S. Günther and S. Sunkle. Enabling Feature-Oriented Programming in Ruby. Technical report (Internet) FIN-016-2009, Otto-von-Guericke-Universität Magdeburg, 2009.

[8] S. Günther and S. Sunkle. Feature-Oriented Programming with Ruby. In *Proceedings of the First International Workshop on Feature-Oriented Software Development (FOSD)*, pages 11–18, New York, 2009. ACM.

[9] R. Hirschfeld, P. Costanza, and O. Nierstrasz. Context-Oriented Programming. *Journal of Object Technology*, 7(3):125–151, 2008.

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

[11] C. Kästner and S. Apel. Virtual Separation of Concerns – A Second Chance for Preprocessors. *Journal of Object Technology (JOT)*, 8(6):59–78, Sept. 2009.

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

[13] G. Kiczales, J. Lamping, A. Menhdhekar, C. Maeda, C. Lopes, J.-M. Loingtier, and J. Irwin. Aspect-Oriented Programming. In *Proceedings of the European Conference on Object-Oriented Programming (ECOOP)*, volume 1241 of *Lecture Notes in Computer Science*, pages 220–242. Berlin, Heidelberg, New York, 1997.

[14] J. Lee and D. Muthig. Feature-Oriented Analysis and Specification of Dynamic Product Reconfiguration. In *Proceedings of the 10th Internationale Conference on Software Reuse (ICSR)*, pages 154–165, Berlin, Heidelberg, 2008. Springer Verlag.

[15] R. E. Lopez-Herrejon and D. Batory. A Standard Problem for Evaluating Productline Methodologies. In *Proceedings of the Third International Conference on Generative and Component-Based Software Engineering (GPCE)*, volume 2186 of *Lecture Notes in Computer Science*, pages 10–24, Berlin, Heidelberg, Germany, 2001. Springer Verlag.

[16] M. Mernik, J. Heering, and A. M. Sloane. When and How to Develop Domain-Specific Languages. *ACM Computing Survey*, 37(4):316–344, 2005.

[17] B. Morin, O. Barais, and J. Jézéquel. K@rt: An Aspect-Oriented and Model-Oriented Framework for Dynamic Software Product Lines. In *Proceedings of the 3rd International Workshop on Models@run.time*, pages 127–136, 2008.

[18] P. Oreizy, N. Medvidovic, and R. N. Taylor. Runtime Software Adaptation: Framework, Approaches, and Styles. In *Companion of the 30th International Conference on Software Engineering (ICSE2010)*, pages 899–910. ACM, 2008.

[19] C. A. Parra, X. Blanc, and L. Duchien. Context Awareness for Dynamic Service-Oriented Product Lines. In D. Muthig and J. D. McGregor, editors, *Proceedings of the 13th International Conference on Software Product Lines (SPLC)*, volume 446 of *ACM International Conference Proceeding Series*, pages 131–140. ACM, 2009.

[20] C. Prehofer. Feature-Oriented Programming: A Fresh Look at Objects. In *Proceedings of the 11th European Conference on Object-Oriented Programming (ECOOP)*, volume 1241 of *Lecture Notes in Computer Science*, pages 419–443, Berlin, Heidelberg, Germany, 1997. Springer Verlag.

[21] M. Pukall, N. Siegmund, and W. Cazzola. Feature-Oriented Runtime Adaptation. In *SINTER '09: Proceedings of the 2009 ESEC/FSE workshop on Software integration and evolution @ runtime*, pages 33–36, New York, NY, USA, 2009. ACM.

[22] M. Rosenmüller, N. Siegmund, G. Saake, and S. Apel. Combining Static and Dynamic Feature Binding in Software Product Lines. Technical Report FIN-013-2009, Otto-von-Guericke-Universität Magdeburg, 2009.

[23] 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*, 11:215–255, 2002.

[24] S. Sunkle and M. Pukall. Using Reified Contextual Information for Safe Run-time Adaptation of Software Product Lines. 2010.

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

[26] J. Withey. Investment Analysis of Software Assets for Product Lines. Technical Report CMU/SEI96-TR-010, Software Engineering Institute, Carnegie Mellon University, 1996.

[27] R. Wolfinger, S. Reiter, D. Dhungana, P. Grünbacher, and H. Prähofer. Supporting Runtime System Adaptation through Product Line Engineering and Plug-in Techniques. In *Proceedings of the Seventh International Conference on Composition-Based Software Systems (ICCBSS)*, pages 21–30. IEEE Computer Society, 2008.