**PAPER SUMMARY**

[S1] The case study project was initiated as part of transitioning the DOPLER tool suite product line from pure academic use to productive use within Siemens [9]. Key project goals were to refactor the tool suite such that it can be used and extended by industrial software engineers and to develop new features necessary to support the industrial use cases.

Overall, we analyzed about 30 person-months of professional development. The observed project provides a unique opportunity to study challenges in product line evolution which is typically difficult as companies are reluctant to allow access to their product lines’ data.

[S2] To realise variability at the code level, product line methods classically advocate usage of inheritance, components, frameworks, aspects or generative techniques. However, these might require unaffordable paradigm shifts for developers if the software was not thought at the outset as a product line. Furthermore, these techniques can be conflicting with a company’s coding practices or external regulations. These concerns were the motivation for the industry–university collaboration described in this paper in which we developed a minimally intrusive coding technique based on tags.

The approach was complemented with traceability from code to feature diagrams which were exploited for automated configuration.

After configuring a product, a feature parser is run over the codebase, which creates a new codebase without the fragments that pertain to non selected features.

[S3] In it, the feature implementation model is introduced as the intermediate level between features and implementation artefacts. The feature interactions are captured in the finer role level, and they help to clarify the complex mapping between features and program implementations. The traceability meta-model for SPL development is introduced and an example on the library management domain is demonstrated.

In this paper, we focus on the visualized representation of the traceability in a software product line, and introduce a comprehensive feature-oriented traceability model. The model explicitly represents the product line artefacts in different abstraction levels. It also contains various kinds of traceability links (explicit/implicit) among the artefacts.

The traceability meta-model for SPL development is introduced and an example on the library management domain is demonstrated.

[S4] In this paper we discussed the proposition of M-SPLearning – a SPL for supporting the development and reuse of mobile learning applications. M-SPLearning was developed throughout a proactive adoption model, according to the basics of SOA. Particularly, we focused on the implementation issues of M-SPLearning, illustrating two pruducts generated from its application in the mobile learning domain. The main contribution of the SPL proposed relies on providing benefits with regard to the overall quality, domain comprehension, and reduction of the time spent in the development and maintenance of mobile learning applications.

To do so, our SPL drew on welldefined data models, leaning on standards such as Java, Representational State Transfer (REST) [25], JavaScript Object Notation (JSON) [26], among others.

The Service module was built in the REST standard, an architectural style that is characterized by the use of Web technologies and protocols for the creation and delivery of services [25]. This style is compatible with the SOA architectural pattern and ensures the availability and consumption of web services in a simple and efficient way.

The Mobile Template module was developed on the Android platform using the Java programming language. This application allows a service, available in Service module, executes a custom build according to the variabilities configured in User Interface module. So, it is possible to generate customized products, which can be installed on Android devices.

[S5] In this paper, we propose a method that addresses these issues by adapting a feature-oriented product line engineering approach. The method is notable in that it guides developers to identify reusable services at the right level of granularity and to map users’ context to relevant service configuration, and it also provides a means to check the validity of services at runtime in terms of invariants and pre/post-conditions of services.

[S6] Many approaches and frameworks have been proposed to realize variability in SOA by applying the concept of Software Product Lines (SPL) where services are the core assets and each member of the service-oriented product line is a possible assembly of those services. However, there are few tools that support these approaches and ease the derivation process of member applications taking into consideration the variability from different perspectives. In this paper we present a tool that facilitates the automatic derivation of SOA applications based on Model Driven Engineering (MDE) as an implementation methodology. The tool is based on the Multiple-Views ServiceOriented Product Line Variability approach. The tool architecture as well as its implemented modules is first described. Then, an example in the e-health domain is presented.

[S7] The work presented in this paper is motivated by the impression that despite the substantial attention in the software product family research community to designing reusable software assets, deriving individual products from shared software assets is a rather time-consuming and expensive for a large number of organizations. In this paper, we have presented a product derivation framework as basis for further discussion. This framework consists of the product family classification mentioned above and a software derivation process that we generalized from practice. The generic derivation process consists of two main phases, i.e. the initial and the iteration phase. In the initial phase, a first configuration is created from the product family assets by assembling a subset of shared artifact or by selecting a closest matching existing configuration. The initial configuration is then validated to determine to what extent the configuration adheres to the requirements imposed by, amongst others, the customer and organization. If the configuration is not deemed finished, the derivation process enters the iteration phase.

[S8] Using the PLE approach we built, at the domain engineering level, a WSbWA-specific lightweight product line architecture and combined it, at the application engineering level, with an Agile Method that uses a domain-specific visual language with direct manipulation and extraction capabilities of web services to perform customization and calibration of a product or WSBWA for a specific customer. To assess the effectiveness of our approach we designed and implemented a tool that we used to investigate the return on investment of the activities related to PLE and AMs. Details of our proposed approach, the related tool developed, and the experimental study performed are presented in this article together with a discussion of planned directions of future work.

In setting up the design of our case study, we asked a group of graduate students who are not familiar with the ongoing study to build, using the .NET Framework©, seven web applications and place them on different web servers in our lab.

[S9] The paper proposes a novel application of product-line model-driven engineering to mobile application development and addresses the key challenges of feature-based native mobile application variants for multiple platforms. Specifically, we deal with three types of variations in mobile applications: variation due to operation systems and their versions, software and hardware capabilities of mobile devices, and functionalities offered by the mobile application. We develop a tool MOPPET that automates the proposed approach. Finally, the results of applying the approach on two industrial case studies show that the proposed approach is applicable to industrial mobile applications and have potential to significantly reduce the development effort and time.

The approach is successfully applied to generate native variants for two industrial case studies: Scramble and Instalapse being developed by our industrial partner. The variants are successfully generated for Android and Windows Phone platforms and are also different in terms of their functionality.

[S10] Most of the current variability resolution tools are based on feature model. However, due to the lack of binding representation in feature model, most of them rely on other build systems for variability resolution to produce product implementations. The disconnection incurs redundant efforts by requiring developers to make or manage two separate artifacts, a variability model and an instantiation script. As a solution for this problem, we previously proposed a software binding application tool, SBAT, which supports build systems and is based on the orthogonal variability description language (OVDL) that simply and strictly defines syntaxes for variability modeling and variability resolution.

we introduced our variability resolution tool, Software Binding Application Tool (SBAT), which is based on the orthogonal variability modeling language (OVDL) [1]. The important aspect of SBAT is that it performs a formally defined binding application operation to achieve variability resolution, where a binding is a set of substitutions and a substitution substitutes a variant for each variation point in a platform artifact (from the domain engineering) to instantiate it to a product artifact (for the application engineering).

Furthermore, for a more practical case study, this paper applies SBAT to a kernel module that is used in both Linux kernel and eCOS, and compares the writing and modifying efforts of applying SBAT with those of applying two tools: Kconfig (for Linux kernel variability resolution) and eCOS-Config (for eCOS variability resolution). As a result, writing a script with SBAT required 26.7% less efforts than doing it with Kconfig and 46.0% less than doing it with eCOS-Config. Also, modifying the script with SBAT required 3.5% less efforts than doing it with Kconfig and 14.2% less than doing it with eCOS-Config.

[S14] Feature Models are insufficient for modeling variability in SOPL, because, services cannot be simply mapped to one or more features, and identification of the mapping depends on knowing the detailed implementation of the services. This research aims at providing an approach to managing the variability in SOPLs so that external services can be involved in the SOPL engineering. This paper presents an overview of the proposal.

In short, the goal of the research is: "To propose a set of methods for managing variability in SOPLs in a way that: 1. external services are included in the model 2. business processes play their role in : a. derivation of service variability model b. instantiation of the service variability model".

This includes methods for: (1) representing the variability needs of services (Service Variability Model, SVM) and automatically extracting it based on business processes, (2) instantiating the SVM, based on specific business process needs, (3) checking the conformance of discovered services with the SVM, and (4) dynamic evolution of the SVM and the service-based applications.

[S15] We propose a feature-oriented solution with aspects for product line architecture design aiming at improving product line architecture evolvability by adopting aspect-oriented techniques, which provide a promising support for modeling crosscutting concerns. Our approach includes guidelines for developing and refining SPL requirements into component-based product line architecture with aspects. We evaluated our approach through a preliminary evaluation which has shown promising results.

[S16] a quantitative case study for evaluating the modularity, complexity, SoC and stability of reusable core assets using distinct technologies for implementing them

[S17] The approach is to model SOA variability with a multiple-view service model and a corresponding meta-model. We integrate SPL concepts of feature modeling and commonality/variability analysis with multiple service requirements and architectural views by using UML and the Service Oriented Architecture Modeling Language (SoaML). The paper describes an automated framework for service-oriented SPL engineering that allows modelers to design, deploy, and execute serviceoriented SPLs

Next, the Code Generation Facility of SoaSPLE was invoked to transform the derived Basic E-Commerce application to Java/Web Services code. Web Services Description Language (WSDL) was produced for the Service Interface View. Business Process Execution Language (BPEL) code was generated for the Service Coordination View. Once code was generated, the Basic E-Commerce member application was deployed and executed using SoaSPLE.

[S18] This paper presents a case study that was performed to evaluate aspect-oriented programming (AOP) as a PL implementation technology. The systematical evaluation is organized along a general evaluation schema for PL implementation technologies.

[S19] This paper presents an industrial experience of such transformation. We propose a non-intrusive **reverse engineering** process for the development of modular services obtained automatically from existing software artifacts, and a variability-driven derivation process to assembly products out of such services. To validate our approach, we have implemented the reverse engineering and derivation processes using real software JEE artifacts from a component framework of reusable functionalities in several different enterprise applications. The results show important benefits in terms of the development time and flexibility.

The product derivation process uses these models to derive a functional product for a given product configuration. The communication between components is implemented via Web Services, REST, or EJB. The results have shown important benefits in the derivation time, and improved flexibility by allowing the distribution of the services in the product with a small overhead in the execution time.

[S20] In this paper, we analyze the insufficiency of two existing solutions in this area and present an architecture-centric approach that meets the requirement. The approach can support product line differences in platforms and functions, and generate both product line code and product code. It is based on a product line implementation mechanism that combines a code generation and separation pattern with an architecture-based code annotation technique. We have implemented the approach, and finished a preliminary evaluation with a chat application.

[S21] ASADAL (A System Analysis and Design Aid tooL) supports the entire lifecycle of software development process based on a PLSE method called FORM (Feature-Oriented Reuse Method) [6]. It supports domain analysis, architecture and component design, code generation, and simulation-based verification and validation (V&V). Using the tool, users may co-develop target software and its test environment and verify software in a continuous and incremental way.

[S22] To implement features, we might use aspect-oriented programming (AOP), in which aspects enable a clear separation between base code and variable code. In this context, recent work provides AspectJ-based idioms to implement flexible feature binding

[S23] UCEd is a use case modeling tool that takes a set of related use cases written in a restricted natural language and automatically generate executable State Charts that integrates the partial behaviours defined by these use cases.

In this paper, we apply an aspect-oriented use case modeling approach to product line system modeling. A use case specification captures stakeholders concerns as interactions between a system and its actors. We adapt our previous work with the introduction of a <> relationship for the expression of variabilities. This relationship is used to model variable and common behaviours across a family of products as use cases. A variability composition mechanism enables building of executable behaviour models for each member of a product line family by integrating common elements with the applicable variable elements.

[S24] This suggests an application synthesis model where configuration and generation are carried out separately by interoperable tools. To this end, we introduce a PLA development toolkit which includes a constraint-based configuration language and an aspect-based generation language supporting the same architecture model. The toolkit imposes dual PLA implementations consisting of a configuration program and a generation program. The compilation of the configuration program yields an interactive configurator used to produce valid configurations at run-time. Valid configurations are then compiled by the generator with the generation program to produce Java applications

[S25] However, as features in general are not independent of each other, changes in the implementation of one feature may cause changes to or side effects in the implementation of other features. We address this challenge by separating feature dependencies from feature implementations using AOP techniques. Specifically, this paper contributes by providing aspect-oriented implementation patterns for feature dependencies (e.g., modification dependency and activation dependency). With the resulting clear separation of dependencies, this approach makes each feature implementation easier to understand and reuse.

In this section, we describe how features and feature dependencies can be mapped into implementation components, such that they can be effectively configured based on the feature configurations of a SPL.

[S26] In this paper, an in-house Software Product Line (SPL), so-called Aurora, is introduced as a platform independent multi-tier Web development environment including the core infrastructure based on Rich Internet Application (RIA) and Enterprise Internet Application (EIA) models.

* Core Banking System (CBS) has been developed in approximately 2 years and completed in 3600 man-months. Java-based Core Banking Application is in operation for more than two years now.
* Central Registry Authority has been developed in one and a half year with 400 man-month efforts and has been in operation in the first quarter of 2005. The system has been developed on Aurora, and it has been integrated with several financial institutions using Web services.
* Insurance Infrastructure is a very recent project being built on Aurora and recently designed middleware RUMBA is being used. The critical issue in insurance domain is the management of dynamically changing complex business rules.

[S27] In this paper, we aim to bridge UML and SPL automatically by having an automatic traslation program. The program will produce UML model based on ABS model, that supports SPL, by using UML-DOP Profile. Besides connecting UML and SPL, the program can also help the developer to achieve coherency between design and implementation. As the results, the UML models produced by automatic translator are represented by XML Metadata Interchange (XMI) documents.

Abstract Behavioral Specification (ABS) is an object oriented modeling language that implements DOP and supports SPL. ABS is an executable modeling language because it is also equipped with ABS tool that can automatically generate running code such as Java, Haskell, and Erlang.

[S28] In this context, this paper presents a repository that allows the management of CBSE and SPL assets and the automated product derivation. Our SPL repository supports registration of several SPLs and their related feature models, reference architectures and components. Furthermore, the proposed work provides an application assembly mechanism where users select features and the repository selects the architecture and the set of assets allowing product derivation. In order to verify the feasibility of our approach, we apply it to the variability management of a Digital TV middleware.

[S29] In this paper, we present our proposed approach to deal with multi-agent systems product lines (MAS-PL) variability management and automatic product derivation. Our approach is implemented as an extension of the GenArch product derivation tool. A case study illustrates how the proposed approach can be used to derive products (instances) from a MAS-PL.

Usan genarch que incluye aspects, pero crean agents con jadex.

[S30] Objective: To address this issue, we propose an approach to developing reliable and maintainable DSPLs in the context of the BSN domain.

Method: Adaptation plans are instances of a Domain Specific Language (DSL) describing reliability goals and adaptability at runtime. These instances are automatically checked for reliability goal satisfiability before being deployed and interpreted at runtime to provide more suitable adaptation goals complying with evolving needs perceived by a domain specialist.

Results: The approach is evaluated in the BSN domain. Results show that reliability and maintainability could be provided with execution and reconfiguration times of around 30 ms, notification and adaptation plan update time over the network around 5 s, and space consumption around 5 MB.

[S31] Agent-Oriented Software Engineering (AOSE) is a software engineering paradigms dedicated to build software applications composed of organizations of agents. Bringing AOSE to the industrial world may prettily benefit from SPL advantages. Using SPL philosophy, a company will be able to define a core MAS from which concrete products will be derived for each customer. This can reduce time-to-market, costs, etcetera. In this paper, we expose the similarities between AOSE and SPL concluding the viability of future research in Multi-Agent Systems Product Lines (MAS-PL).

In Figure 3, we show the hierarchical goal diagram of our case study using TROPOS.

[S32] Current tools and languages, however, force a programmer to decide between static and dynamic composition during development. In this paper, we present an approach that employs code generation to support static and dynamic composition of features of a single code base. We offer an implementation on top of FeatureC++, an extension of the C++ programming language that supports software composition based on features. To simplify dynamic composition and to avoid creation of invalid products we furthermore provide means to (1) validate the correctness of a composition at runtime, (2) automatically instantiate SPLs in case of stand-alone applications, and (3) automatically apply interaction code of crosscutting concerns.

[S34] In this paper, we show that **change propagation probability** (CP) is helpful and effective in assessing the design quality of software architectures. We propose to use the CP to assess the evolution of the architecture of software product lines through different releases. We use CP to investigate whether aspect oriented SPL has better maintainability evolution than object-oriented SPL.

Change propagation probability CP= [CPij] for an architecture is the conditional probability that a change originating in one component of the architecture requires changes to be made to other components.

We found that AO SPL has better maintainability compared to 00 SPL estimated using the change propagation coefficient across the different releases of the case study.

[S35] In this paper, we have adopted a component implementation model called COSMOS [19,20], which is a representative component model [4],

Objective: This paper evaluates the positive and negative change impact of component and aspect based design on PLAs. The objective of the evaluation is to assess how aspects and components promote PLA stability in the presence of various types of evolutionary change. To support a broader analysis, we also evaluate the PLA stability of a hybrid approach (i.e. combined use of aspects and components) against the isolated use of component-based, OO, and AO approaches.

Results: The combination of aspects and components promotes superior PLA resilience than the other PLAs in most of the circumstances.

[S36] A particular domain where the adoption of such approach may bring relevant benefits is the mobile game domain given the big diversity of handsets and the large number of commonalities among these games. However, applying SPL approaches in such domain is not trivial because of some restrictions, such as reduced memory and application size. In this context, this work presents a practical approach to implement core assets in a SPL in the mobile game domain combining good practices from previous work and briefly describing a case study performed with three mobile games.

Screenshots of the games in case study. A) Monga. B) American Dad – Roger’s Escape. C) Zaak. D) Smart Escape. Considering specifically theComponent Complexitymetric value, we also realized that it may be slightly affected due to the number of conditional compilation tags.

[S37] To provide more flexibility for implementing software product lines, we propose **delta-oriented programming** (DOP) as a novel programming language approach. A product line is represented by a core module and a set of delta modules. The core module provides an implementation of a valid product that can be developed with well-established single application engineering techniques. Delta modules specify changes to be applied to the core module to implement further products by adding, modifying and removing code. Application conditions attached to delta modules allow handling combinations of features explicitly. A product implementation for a particular feature configuration is generated by applying incrementally all delta modules with valid application condition to the core module. In order to evaluate the potential of DOP, we compare it to FOP, both conceptually and empirically.

, the JAK language, a superset of JAVA containing constructs for feature module refinement.

Graph Product Line (GraphPL), suggested in [24] as a benchmark to compare SPLs architectures

[S38] The trends toward product line development and toward adopting more **third-party software** are hard to combine. The reason is that product lines demand fine control over the software (e.g., for diversity management), while third-party software (almost by definition) provides only little or no control. A growing use of third-party software may therefore lead to less control over the product development process or, vice-versa, requiring large control over the software may limit the ability to use third-party components. Since both are means to reduce costs and to shorten time to market, the question is whether they can be combined effectively. In this paper, we describe our solution to this problem which combines the Koala component model developed within Philips with the concept of build-level components. We show that by lifting component granularity of Koala components from individual C files to build-level components, both trends can be united. The Koala architectural description language is used to orchestrate product composition and to manage diversity, while build-level components form the unit of third-party component composition.

Koala [10,11] is a component model consisting of an architectural description language (ADL) and tool support.

The first is GEARS [7]. This is a software product line development tool, which explicitly supports the integration of existing (i.e., unchanged) software. This implies that GEARS can deal with software over which no control exists.

Nix is a promising tool for safe software deployment. Underneath is a functional language that provides advanced diversity features. Nix was explicitly designed for managing diversity and safe orchestration of open source software composition.

Configurations In order to build a product from a set of components, a configuration has to be defined. A configuration is a component definition without provides or requires interfaces. It must bind all unbound requires interfaces of its constituent component instances. In the example above, this means binding the rFoo and rBar interfaces.

[S39] 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.

**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.

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.

The applications are becoming an abstraction which enables **fine-grained modifications** and runtime adaptation

[S40] In this chapter we describe the product line models, and show how to apply them for developing and evolvin**g Web products**. A product line captures the common and variable aspects of software systems as key assets under a common architecture. Software companies are increasingly adopting this approach in order to accelerate the development of families of similar software products. In certain domains, such as the Web systems, development and maintenance operations are required more often. New techniques to engineer Web sites are needed in order to reduce the time to market for the Web products and to maintain the systems afterward. The authors believe that understanding the notion of lightweight product line and the role that the architecture plays will help software engineers in the construction of software products, and they will be able to manage the evolution effectively against future changes.

In this section we will study how product lines can be successfully employed in the development and maintenance of Web systems.

First we have developed the 70% of Web product (i.e., the environmental observatory) from scratch and without using any product line approach. This took nearly 230 effort hours. Later we decided to engineer the same product using reusable components built in an informal way. These reusable components were obtained from the experience gained in the development of the e-learning platform, which has several similarities to the environmental observatory. The development of the reusable assets took over 140 effort hours. Afterwards, we have engineered again the environmental observatory, but with taking advantage of such reusable assets. We have spent 120 effort hours building the same product with reuse; thus the effort reduction is evident. Finally, we have built the product using our lightweight product line model. Establishing the PLA took around 130 effort hours and the construction of the core assets under the product line was done in 120 effort hours. At last, the product was engineered in only 66 effort hours. Therefore, we can see a significant reduction in the time employed to engineer the environmental observatory compared to the other two approaches. Of course we have to employ time for setting up the product line and for developing the core assets but as we build more products, the product line will be more profitable.

Later we decided to engineer the same product using reusable components built in an informal way.

[S41] In this paper we show that adding new features to a product line over time results in crosscutting changes to a system and its constituting components. Given the nature of problems experienced when evolving consumer products with new features, we outline opportunities for using aspect-oriented technologies to address some of these problems. We use the Philips TV product line architecture, as an example embedded commercial product line, to illustrate the evolution problems and outline the possible ways of addressing those using aspects

So far, several ”flavors” of aspect weavers for C language have been developed. These include AspectC++ [8], Aspicere [3], C4 [4] , WeaveC [5], and AspectC [2].

In this paper we proposed several dimensions where aspects could ease the evolution problems in today’s componentbased product lines. To date, to the best of our knowledge, there is no approach or a tool that would scale up to a commercial use of aspects for embedded consumer products, and that addresses all levels where aspect support could be used to facilitate adding aspects in existing commercial products

[S42] This paper reports a quantitative study that evolves two SPLs to assess various design stability facets of their aspect-oriented implementations. Our investigation focused upon a multi-perspective analysis of the evolving product lines in terms of modularity, change propagation, and feature dependency. We have identified a number of scenarios which positively or negatively affect the architecture stability of aspectual SPLs.

This study evolved two reallife SPLs in order to assess the capabilities of AOP mechanisms to provide SPL modularity and stability in the presence of realistic change tasks. Such evaluation included three complementary analyses: implementation modularity, change propagation and feature dependency.

From this analysis we discovered a number of interesting outcomes. Firstly, the AO implementations of the studied SPLs tend to have more stable design particularly when a change targets optional and alternative features (Section 4.2 and 5.1). This indicates that aspectual decompositions are superior in those situations, especially when considering the Open-Closed principle [20]. However, AO mechanisms do not cope with the introduction of widely-scoped mandatory features or when changing a mandatory feature into alternatives (Section 4.1 and 5.1). Furthermore, such mechanisms usually scale well for dependencies that do not involve shared code, although AspectJ faces difficulties to address different SPL configurations.

[S43] We developed a customizable Eclipse extension for SPLE that consists of around 38 plug-ins. The resulting tool, called EASy-Producer, extends the Eclipse IDE by the capability to support the creation and management of software product line projects. To provide this capability, EASy-Producer utilizes the extension concepts of the Eclipse platform and integrates additional frameworks, like Xtext.

SimElevator SPL. This SPL provides different variants of an elevator simulator implemented in Java

All three languages (IVML, VIL, VTL) are custom-developed DSLs.

VM consists of four main VM Activities (VMA) [6]:

VMA 1: Modeling variability

VMA 2: Modeling the relation between variability and generic artifacts

VMA 3: Describing the configuration of an instance

VMA 4: Deriving customized products

[S44] The feature implementation model captures feature interactions (including cross-cutting interactions) in the finer role level, and help to clarify the complex mapping between feature and program implementation. So, feature-driven program-level customization and configuration can be enabled by the model and role instantiation. AOP (Aspect-Oriented Programming) is adopted as the implementation technology for product derivation on the program level. Then program-level composition can be implemented by aspect weaving to finally achieve the feature-driven product derivation.

In this paper, we propose a product derivation method in which feature-driven program-level customization and composition are supported by feature implementation modeling, instantiation and aspect weaving in AOP. The main contribution of this paper is enabling feature-driven program-level customization and composition for product derivation by introducing an intermediate feature implementation model between feature model and program implementation along with corresponding customization and instantiation. However, our method doses not cover the issue of featuredriven DSSA (Domain Specific Software Architecture) design. In fact, domain-level design and implementation are assumed to have been done. Our method provides a mechanism of feature implementation design and instantiation to map feature-level customization to program-level configuration and composition. It is an implementation technology for product derivation in SPL. In the future research, we will focus on more systematic and comprehensive support for feature-driven implementation design and SPL evolution management.

[S45] Due to new computing trends such as ubiquitous computing, the complexity of software design is leading to increased effort for development, maintenance and configuration. One promising way to address these issues is self-adaptation. Self-adaptive systems are capable of changing their own behaviour to adapt dynamically to context changes at runtime. Among the recent development approaches available to achieve adaptation in software systems, dynamic software product lines attempt to face challenges of runtime variability mechanisms. Hence, we propose a behavioural adaptation method based on software product line and the MAPE-K feedback loop. The objective is to determine a valid SPL configuration (i.e. a set of features to activate and deactivate) with respect to a variability model for a given context.

[S46] In this paper, we describe how feature modeling can be integrated with aspect-oriented programming to perform automated product derivation efficiently and effectively in the context of large-scale product lines.

[S47] This paper introduces rbFeatures, an implementation of first-class features in the dynamic programming language Ruby. Our goal is to show how such a language extension works with respect to its dynamic host language and the applicability of our results. In particular, we present a step-by-step walkthrough how to use rbFeatures in order to implement known case-studies like the Graph Product Line or the Expression Product Line. Since we created a pure Ruby language extension, rbFeatures can be used with any existing programs and in any virtual machine implementing Ruby.

This configuration can be represented as a class containing activation statements

The dynamic nature of Ruby allows two kinds of usages: static configuration of a variant before executing the program and dynamic configuration at runtime

[S48] However, currently there is no dedicated method for proactively developing SPLs using design patterns suitable for realizing variable functionality. In this paper, we present a method to perform generative SPL development with design patterns. We use role models to capture design patterns and their relation to a variability model. We further allow mapping of individual design pattern roles to (parts of) implementation elements to be generated (e.g., classes, methods) and check the conformance of the realization with the specification of the pattern. We provide definitions for the variability-aware versions of the design patterns Observer, Strategy, Template Method and Composite. Furthermore, we support generation of realizations in Java, Cþ þ and UML class diagrams utilizing annotative, compositional and transformational variability realization mechanisms. Hence, we support proactive development of SPLs using design patterns to apply best practices for the realization of variability. We realize our concepts within the Eclipse IDE and demonstrate them within a case study.

**Transformational approaches** designate a core variant that is changed through adding, modifying and removing elements to create a target variant that conforms with a particular configuration, e.g., as with delta modeling.

In our future work, we will further explore how to reduce specification effort for design patterns to increase the benefits of our approach

[S49] They are often described using feature models which, as we proposed in a previous work, can be built from possibly incomplete, documented UML use case diagrams assets using the Formal Concept Analysis method, semantic model and trigger model. In order to evaluate this approach, we present in this paper the UC2FM-tool which automates all its steps. In addition, we report on a comparison of the values of quality metrics of feature models produced by our approach with those of existing feature models built by experts for five different domains.

UC2FM-tool (Use Cases to Feature Model tool).

The approach shows that FMs can be built automatically not only from source codes, but also from descriptions and UC diagrams. It uses four possible types of product variant assets: complete and documented UC diagrams, incomplete but documented UC diagrams UC diagrams with no scenarios, and/or just scenarios written in natural language.

Furthermore, on the theoretical level, we are currently examining how to make better use of the semantic information present in the UCs scenarios in order to derive the code skeleton of each feature in the FM.

[S50] Its implementation relies on three pillars: an embedded real-time framework, extensive code generation and a set of design and coding patterns. This article describes these principles that determine the NH90 SPL architecture, the techniques used for code generation, and a selection of the design and coding patterns, as an example of a software product line implementation in a real project within the avionics domain.

The NH90 is a medium weight multi-role military helicopter that comes in two basic versions: the Tactical Transport Helicopter (TTH) and the NATO Frigate Helicopter (NFH). It is being produced in 23 variants for 14 nations and their armed forces who ordered a total of more than 500 NH90s

In order to cope with the high number of software variants and technology variations, the NH90 software team developed concepts and strategies for software architecture and tool modifications based on Software Product Line (SPL) principles [1].

More than 50 percent of the adaptation source code of the main computers is generated from data maintained in an ORACLE database called ODIN. ODIN is an acronym for OFRS Data and Interfaces in NH90

Despite of these challenges we believe that product lines are an promising strategy to cope with the incessantly growing complexity of modern military avionics systems.

[S51] We present a Software Product Line Infrastructure (SPLI) that has been designed to increase the reuse of software efforts in product populations. The SPLI takes a bottom-up approach by structuring product features in highly reusable software components called Active Components which contain different types of artefacts. Variability is expressed using domainspecific models and formal variability models. Variability is bound during product derivation by executing model-toartefact transformations. Components are active because they are invoked during the derivation process, thereby empowering the component.

Compared to a software product family, a software product population [18] encompasses an even more diverse set of software products; a family of software product families. **Product populations can be seen at software vendors that develop archetypical types of software systems for a wide range of markets**.

The increased product line scope when dealing with a software product population leads to extra complexity. The design of the SPLI is based on the premise that this idea of divide and conquer is essential to manage the involved complexity when a wide variety of software products must be supported by the software product line.

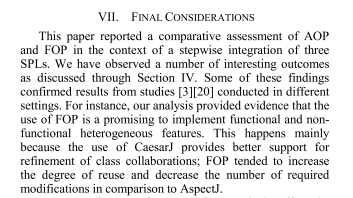
Software efforts reuse in software product populations is improved in two ways: 1) an Active Component can be reused in the wider software product spectrum as variability is increased by using domain-specific models to express variability and executing model-to-artefact transformations to derive implementations and 2) application design is reused by using model specialization and allowing step-wise refinements of applications

Active Component can provide both implementation (e.g. source-code, libraries, generators) and/or design (e.g. entity models, state machine models, variability models).

[S52] In particular, we propose new extensions to our existing model-based product derivation tool, called GenArch, in order to address the new abstractions and mechanisms provided by the Spring and OSGi component models. The GenArch extensions enable the automatic instantiation of product lines and applications - implemented using these component technologies. Moreover, it also enables different levels of customization, from fine-grained configuration of component properties to the automatic selection of components that will compose the final product

In this paper, we presented two extensions to a product derivation tool which address the integration of Spring and OSGi mainstream component-based technologies.

[S53]



[S56] In this paper, we present the design and implementation of a transformational model of a product line of scalar vector graphics and JavaScript applications. We explain how we simplified our implementation by lifting selected features and their compositions from our original product line (whose implementations were complex) to features and their compositions of another product line (whose specifications were simple). We used operators to map higher-level features and their compositions to their lower-level counterparts. Doing so exposed commuting relationships among feature compositions in both product lines that helped validate our model and implementation.

[S57] This paper presents a transformative code generation technique for the static optimization and customization of embedded software. The approach supports the development of product families by separating core functionality from variable facets. The implementation technique utilizes generative programming techniques in order to minimize runtime memory requirements and maximize performance within an embedded environment.

[S58] In software product line engineering, core assets are shared among multiple products. Core assets and products generally evolve independently. Developers need to capture evolution in both contexts and to propagate changes in both directions between the core assets and the products. We propose a version control system to support product line engineering by supporting the evolution of product line, product derivation, and change propagation from core assets to products and vice versa.

We have described an approach that is capable of versioning multiple types of product line projects including document product lines. It has a version model for a product line consisting of a single core assets project and multiple product projects where core assets are shared among the products through the use of shared components. Using the shared component data structure and the branching of the core assets project, we are able to support independent development of core assets and products and change propagation between them.

[S59] In this paper we describe the role of aspects in software product lines. According to the role of aspects, we classify the features and propose a process to map features to architectural components in an Aspect-Oriented Product Line Engineering (AOPLE).

Sochos et al. [3] proposes the Feature–Architecture Mapping (FArM) as a way to progressively transform a feature model into architectural component model.

[S60] This paper presents a model-based generative approach to mapping features to aspects across different artifacts of an product line. Our main aim is to enable the smooth and systematic derivation of aspect-oriented software family architecture. Our approach is complementary to a set of previously-proposed modularization guidelines to implement aspect-oriented frameworks. We present details about the suite of mappings supported by our generative model, illustrate them in heterogeneous case studies, and discuss several implementation issues for its accomplishment.

In our particular tool implementation, we are using **JET** (Java Emitter Templates) to codify our templates. JET is the template engine of the Eclipse Modeling Framework (EMF) plugin. It can be used to implement templates for any kind of implementation element (classes, aspects, configuration files).

[S61]Our objective is to create a model-driven product line for SCM systems. By explicitly describing the different concepts using models, reuse can be performed on the modeling level. Since models are executable, the need for manual programming is eliminated. Furthermore, by providing a library of loosely coupled modules, we intend to support flexible composition of SCM systems.

Furthermore, we developed an executable object-oriented domain model which was annotated with features from the feature model. A specific SCM system is configured by selecting features from the feature model and elements of the domain model realizing these features. Results: Due to the orthogonality of both feature model and domain model, a very large number of SCM systems may be configured. We tested our approach by creating instances of the product line which mimic wide-spread systems such as CVS, GIT, Mercurial, and Subversion

Model-driven software engineering [4,5] puts strong emphasis on the development of high-level models rather than on the source code. Models are not considered as documentation or as informal guidelines how to program the actual system. In contrast, models have a well-defined syntax and semantics. Moreover, model-driven software engineering aims at the development of executable models. Ideally, software engineers operate only on the level of models such that there is no need to inspect or edit the actual source code (if any).

FeaturePlugin [21] and Fujaba [22] were used for feature modeling and creating the executable domain model, respectively.

FeaturePlugin is a stand-alone tool which supports cardinality-based feature modeling [17]. The tool provides a tree editor for creating a feature model and an interactive configurator for deriving a consistent feature configuration from the model. Fujaba [22] is an object-oriented CASE tool for model-driven software engineering. Its added value compared to other CASE tools is provided by story diagrams and the ability to create fully executable source code out of them. In contrast to most other tools, Fujaba does not merely generate implementation frames for method bodies. Rather, complete executable Java code is generated from a story diagram acting as a behavioral model of a method defined in a class diagram

In order to combine feature modeling and domain modeling, we developed a tool called MODPLFeaturePlugin [19,20]. Using this tool, the user maps features to corresponding domain model elements. Furthermore, a large amount of consistency and plausibility checks are available which help to ensure the syntactical correctness of the resulting configurations. Furthermore, automatic repair actions propagate feature annotations to depending model elements

USAN PACKAGE DIAGRAM Y ANOTATIONS.

Software Configuration Management (SCM) denotes the discipline of controlling the evolution of large and complex software systems. Over the years a wide variety of different tools and systems have been developed. They comprise small sized ones, like RCS [6], medium-sized systems like CVS [7] or Subversion [8] and even largescale industrial tools such as Adele [9] and ClearCase [10].

[S62] We present a framework and related tool suite for modeling and managing the variability of Web service-based systems for design and run-time, respectively. It is an extension of the COVAMOF framework for the variability management of software product families, which was developed at the University of Groningen. Among the novelties and advantages of the approach are the full modeling of variability via UML diagrams, the run-time support, and the low involvement of the user. All of which leads to a great deal of automation in the management of all kinds of variability.

COVAMOF is a variability management framework, developed at the University of Groningen, to handle the issues in variability management relevant for the software industry (Deelstra et al., 2005; Sinnema et al., 2004, 2006a,b). It offers facilities to model the variability in a software system over multiple layers of abstraction. The COVAMOF framework is designed specifically for use with software product families

he COVAMOF framework helps developers in deriving these individual products by providing an associated tool suite, called COVAMOF-VS, which is an add-in for Microsoft Visual Studio .NET

The newly developed language, called VxBPEL, has extra XML elements to support variation points and variants in a BPEL process.

BPEL is an XML-based programming language that can be used to describe the interaction between Web services at the message level; in this way it also describes their composition.

To model **variability in the architecture of Web service-based systems** we first define a profile for the Unified Modeling Language (UML). This profile allows us to model COVAMOF variability concepts in individual UML diagrams, and to make variation points span over multiple UML diagrams.

Class diagrams

Activity diagrams

Sequence diagrams

Deployment diagrams

[S63] **Dynamic Software Product Lines (DSPL)** engineering has emerged as a promising strategy to develop Software Product Lines (SPL) that incorporate reusable and dynamically reconfigurable artifacts. The central purpose of DSPL is to handle adaptability at runtime through variability management, as well as to maximize the reuse of components.

In this paper, we report on an exploratory study aimed at evaluating the objectoriented and aspect-oriented solutions on DSPL evolutionary scenarios. In this empirical evaluation, the aspect-oriented solution yielded better results in terms of measurements such as Weighted Operations per Component (WOC), Lines Of Code (LOC), Lack of Cohesion Over Operations (LCOO), Coupling between components (CBC), and Response For a Class (RFC). The use of aspects indicates that it provides assets with lower complexity, lower coupling, and higher cohesion.

In order to increase the reliability of measures, the **AOPMetrics tool** was used to collect the data set that was evaluated in this exploratory study.

[S64] In this paper we exploit the potential of independent and pluggable software units (PUs) for assembling the products of a related family. PUs support application variability through Inheritance of Topology (IT), a construct that reuses the topology of PUs network models. IT enables new applications to be derived from existing ones by just requiring the definition of the differences. We present results from the development of a microwave oven SPL.

PUs communicate only through input and output gates becoming thus independent of the context they can be used. JUSE provides an implementation of PUs in the Java/Groovy language

We introduce Composition Diagrams (CDs) as a means for describing families of products. CDs provide a comprehensive representation for all the applications that can be created from a product line

JUse, an implementation of pluggable software units in the Java/Groovy language.

[S65] This paper presents quantitative and qualitative analysis of how feature modularity and change propagation behave in the context of two evolving SPLs, namely WebStore and MobileMedia. Quantitative data have been collected from the SPLs developed in three different variability mechanisms: FOP refinements, conditional compilation, and object-oriented design patterns. Our results suggest that FOP requires few changes in source code and a balanced number of added modules, providing better support than other techniques for non-intrusive insertions. Therefore, it adheres closer to the Open–Closed principle. Additionally, FOP seems to be more effective tackling modularity degeneration, by avoiding feature tangling and scattering in source code, than conditional compilation and design patterns. These results are based not only on the variability mechanism itself, but also on careful SPL design. However, the aforementioned results are weaker when the design needs to cope with crosscutting and fine-grained features.

Listing 1. Example of variability management with conditional compilation.

[S66] Although feature models play an important role in describing SPL elements, they are limited to provide high-level feature decompositions that do not explicitly represent the SPL architecture. To tackle this problem, we present PL-AspectualACME, an extension of the ACME architecture description language that enriches existing abstractions to express architectural variabilities. They support the specification of product variations without forcing architects to learn many new abstractions. We evaluate the applicability of our proposal in the context of a real large-scale system, the Ginga SPL architecture.

In fact, SPL architects usually employ a component-based design approach in order to promote a smooth mapping of their decisions to elements of the implementation platform

Architectural description languages (ADLs) typically use architectural styles to define vocabularies of types of components, connectors, properties, and sets of rules that specify how elements of those types may be legally composed in a reusable architectural domain. AspectualACME is an aspect-oriented (AO) extension of ACME [6], a general-purpose ADL proposed as an architectural interchange language.

[S67] Features are separated in models and composed by aspect-oriented composition techniques on model level. Model transformations support the transition from problem to solution domain. Aspect-oriented techniques enable the explicit expression and modularization of variability on model, code, and template level. The presented concepts are illustrated with a case study of a home automation system.

**Model-driven software development (MDSD)** [4] improves the way software is developed by capturing key features of a system in models which are developed and refined as the system is created. During the system’s lifecycle, models are synchronized, combined and transformed between different levels of abstraction and different viewpoints. In contrast to traditional modelling, models do not only constitute documentation but are processed by automated tools. Thus models have to be formal, whereas every model is an instance of a meta model. The meta model defines the vocabulary and grammar, i.e. the abstract syntax, used to build models. In order to be useful for MDSD, models have to be complete regarding the abstraction level or viewpoint they describe. **A Domain Specific Language (DSL)** [4] is a formalism for building models: It encompasses a meta model as well as a definition of a concrete syntax that is used to represent the models. The concrete syntax can be textual, graphical or using other means, such as tables, trees or dialogs. Different DSLs can use the same meta model while varying in their concrete syntax. The models built with these DSLs will look different, but will all have the same meaning. The meta model is what the tools care about, whereas the concrete syntax is what the DSL users care about. It is essential, that the concrete syntax can sensibly represent the concepts the DSL is intended to describe.

Negative variability selectively takes away parts of a creative-construction model based on the presence or absence of features in the configuration models.

The second alternative uses positive variability. We start with a minimal core and selectively add additional parts

Another implementation uses the CAM-DAOP aspect-oriented component infrastructure [20].

**TOOLS**

There are also tools for building custom textual editors including syntax highlighting, custom outline views and code completion, EMF-based examples include oAW’s xText [23] and INRIA’s TCS [28].

For mapping problem domain models to solution domain models we need M2M transformation languages and tools. Although these kinds of languages are a relatively new breed, there are solutions available today. Based on Eclipse, there is INRIA’s ATL [29] and oAW’s Xtend [23]. As part of the Eclipse M2M project, a QVT [14] implementation will be provided in due course [30]. Different template languages have different levels of support for variability. oAW’s Xpand [23] supports AO on templates.

[S68] In this position paper we try to improve the development and evolution of service oriented systems. To achieve this we propose a modification of the traditional product line approach to support composite services. We use a lightweight product line with specific variability information to facilitate web service composition as well as the evolution of such systems.

The composition between services can be specified using different languages (e.g.: WSFL, BPML, BPEL).

[S69] In this paper we propose a methodological approach and an implementation framework, based on a plugin component-based development, which allows us to move from an architectural specification of the SPL to its implementation in a systematic way. We show the suitability of this framework through its application to the TOOL•one case study SPL.

a PLA precisely captures, in a single specification, the overall architecture of a suite of closely-related products [8]. A PLA specification focuses on modeling mandatory elements (which are present in the architecture of each product), optional elements (which may or may not be present in a product), variant elements (which can be mandatory or optional, and can be chosen to be one of a number of different alternatives) [14] (see Figure 1, top-left portion), and explicitly models connection and communication constraints.

Koala, instead, is an ADL specially designed for modeling embedded software for consumer electronics

M´enage [24] is an environment for managing evolving architectures in SPL. It provides a definition of how to represent PLAs (addressing variabilities and also evolution) and a graphical environment which facilitates the specification, in the defined representation, of PLAs.

KobrA (Component-based Application Development) [5] is a software processand product- centric engineering approach which uses a component-based approach in each phase of the development process

Pure::Variant

In [10], Cortellessa et al. faced such a problem by proposing TOOL•one, a framework for integration of functional and non-functional analysis

Other tools proposed by the SPL community include Holmes [21], which supports SPL development and whose architecture is based on JavaSpaces, PuLSEBEAT [19] which supports all steps of the scoping process, and [17] which supports top-down as well as bottom-up traceability in product families. None of such approaches and tools seem to support implementation of SPLs. For sake of completeness, somehow related to our topic are also Koala [25], xADL [4](with its C2 framework) and DiscoTect [26] (for architecture recovery).

[S70] This paper introduces Aspect-Oriented Programming (AOP) as the method for the improvement of the assembling process in software product line. The method that assembles core asset and variabilities is described by grammar elements such as Join point, pointcut and advice without code-change. We analyze the requirements of the mini-system as an example adapting AOP and show its practicality by the implementation of Aspect-Oriented language, AspectJ

AspectJ is Java programming language plus aspectoriented concept. Grammatical constructs to present crosscutting concerns in AspectJ are dynamic crosscutting concern that adds or modifies behavior of a program and static cross-cutting concern that intervene in modifying static structure of a program.

[S72] In this article, we share experiences from the ongoing incremental adoption of explicit variability management at TRW Automotive’s department for automotive slip control systems—located in Koblenz, Germany. On the technical side, the three key drivers of this adoption effort are (a) domain modeling and scoping, (b) handling of variability in requirements and (c) tighter integration of software engineering focus areas (e.g., domain modeling, requirements engineering, architectural modeling) to make use of variability-related data. In addition to implementation challenges with using and integrating concrete thirdparty tools, social and workflow-related issues are covered as well. The lessons learned are presented, discussed, and thoroughly compared with the state of the art in research.

In this article,1 we particularly focus on handling variability in domain modeling and requirement engineering, because we deem good engineering and preparation in the problem space to be the key to successful reuse during development activities in the solution space.

Concerning tools, pure::variants5 is used for variant management, DOORS6 for requirements engineering and management, and Rhapsody7 for architecture modeling

pure::variants The tool pure::variants is being developed by pure-systems and is used for modeling features, expressing product variants in terms of features, and generating tailored artifacts. The tool ships with various connectors (also called bridges) to other thirdparty tools, such as DOORS and Rhapsody. As it is based on the well-known Eclipse platform, it can be extended by writing plug-ins in the Java programming language.

• DOORS The tool DOORS is being developed by IBM Rational (originally by Telelogic) and offers a client– server system for requirements engineering and management. Requirements are stored in modules that are represented like tables in a relational database.

• Rhapsody The tool Rhapsody is being developed by IBM Rational (originally by I-Logix and later by Telelogic) and supports system engineers with modeling static and dynamic aspects of software systems using UML and SysML. Code generation for different languages is supported as well as the simulation of behavioral models such as state charts.

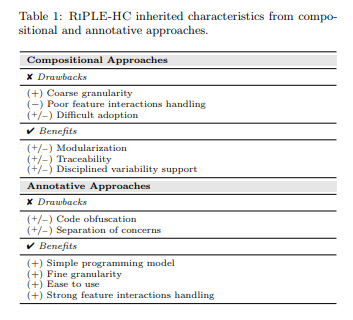
[S73] This paper presents the RiPLE-HC, a strategy aimed at blending compositional and annotative approaches to implement variability in Javascript-based systems. Results. We applied the approach in an industrial environment and conducted an academic case study with six open-source systems to evaluate its robustness and scalability. Additionally, we carried a controlled experiment to analyze the impact of the RiPLE-HC code organization on the feature location maintenance tasks. Conclusion. The empirical evaluations yielded evidence of reduced e↵ort in feature location, and positive benefits when introducing systematic reuse aspects in Javascript-based systems.

The RiPLE-HC is the RiSE Product Line Engineering approach for Hybrid Composition of JavaScript-based systems.

RiPLE-HC deals with the use of preprocessor directives (annotative approach) to manage variability at implementation level. IF DEF

The RiPLE-HC relies on the FeatureIDE capabilities to automatically create the containment hierarchy (Fig. 1), in which there is a directory to store all the code belonging to each concrete feature.

We planned and carried out a controlled experiment with Software Engineering students to gather evidence on the maintenance e↵ort demanded by the RiPLE-HC featurebased code organization in comparison to the current stateof-the-practice of JavaScript code organization



[S74] In this paper, we present a multi-tenant single instance SaaS evolution platform based on Software Product Lines (SPLs). The platform specifies a set of evolution rules, based on feature modeling, that govern evolution decisions. We also present the early implementation phases of the proposed approach based on SPLs and Model Driven Architecture (MDA) concepts.

Software-as-a-Service (SaaS) has recently been adopted by many organizations, tenants, to get their work done through subscription-based services. To leverage economies of scale, SaaS applications are provided to multiple tenants sharing the same software and hardware resources based on virtualization concepts. This can be referred to as multi-tenant applications (MTAs). Because tenants subscribing to a SaaS application have varying requirements, a one-size-fits-all SaaS that provides the same set of services for all tenants is not an efficient solution.

MDA [5] is an approach adopted by the Object Management Group (OMG) to provide reusable solutions addressing the whole development lifecycle. MDA relies on the use of models, with various abstraction levels, to carry out the different phases of system development. MDA captures the system requirements independently from any platform specific details using a platform-independent model (PIM). Then, it integrates the platform-specifications to a PIM to generate a platform-specific model (PSM). Model-to-Model (M2M) transformations are used to transform one model to another based on a set of predefined rules.

All M2M transformation carried out by the framework are based on Query-View-Transformation (QVT3 ) which is a standard language adopted by the OMG and used to carry out transformations between models

The PIMs generated by the platform are represented as XMI4 files. An XMI file consists of nested tags that correspond to features and feature mappings.

[S75] This paper presents a new approach to delta modeling and a supporting tool suite: the abstract notion of a delta is refined to be a consistency-preserving edit script which is generated by comparing two models. The rich structure of edit scripts allows us to detect conflicts and further relations between deltas statically and to implement restructurings in delta sets such as the merging of two deltas. We illustrate the tooling using a case study.

Conventional SPLE methodologies assume “a product” of the SPL to be a source program which is compiled to an executable program. However, source code has been replaced by models in various domains where model-driven development (MDD) methods are used [2], [23], e.g. automotive systems and embedded software. Modeling languages such as Simulink or subsets of the UML are used here. Generating a product during application engineering thus means generating a model which is complete and executable.

Usually, delta-modules must be manually written using a specific textual delta-language which can be considered as DSL for a specific modeling domain <http://pi.informatik.uni-siegen.de/Projekte/sipl/>

[S76] However, one of the challenging problems for the realization of this vision is an need for design and management of variants of SOAbased solutions. Such SOA-based solutions require customization to meet stakeholders’ individual functional and non-functional requirements. In this chapter, a methodological foundation for modeling and developing variant-rich SOA-solutions by incorporating the principles of Software Product Line Engineering (SPLE) into the SOA development life cycle

That is, a reference business process model is a union of all the business processes of the product line. It provides the common business logic for orchestration and choreography of services, which implement features. The reference model comprises functional interfaces specifying services capabilities, pre and post conditions of the services, and configuration properties representing the data needed to configure a service before its use, and service bindings. The reference business process model can be modeled by using processoriented modeling languages (e.g., BPMN, EPC, and/or YAWL), and incrementally refined and optimized

VxBPEL is proposed as an extension of Business Process Execution Language (BPEL) for to the process description and definition

VxBPEL builds upon COVAMOF [50], a framework for modeling variability.

Service-Oriented Modeling and Architecture (SOMA) proposed by IBM [2] has been developed as a generic development method for SOA-based applications. SOMA provides the guidelines for identification and specialization of services that realize and implement business processes through service composition

Service-Oriented Modeling and Architecture (SOMA) proposed by IBM [2] has been developed as a generic development method for SOA-based applications. SOMA provides the guidelines for identification and specialization of services that realize and implement business processes through service composition

[S78] This paper presents the Software Product Line Integration Tool (SPLIT), our strategy to such transformation in Heinsohn Business Technology (HBT). We propose a non-intrusive reverse engineering process for the development of modular services obtained automatically from existing software artifacts, and a variability-driven derivation process to assembly products out of such services. To validate our approach, we have implemented and tested SPLIT using real software artifacts from a framework of reusable components for several enterprise applications. The results show important benefits in terms of the development time and flexibility.

Currently we support three types of bindings for the services (EJB, WebServices and REST).

Spoon is a tool that analyses and transforms Java code using processors. A processor is a Java class that allows developers to look for and modify elements of an application with a high level of granularity. Spoon creates an abstract syntax tree of the code being analyzed and offers the API to navigate through the tree and eventually perform modifications. EMF on the other hand provides the functionalities to create and manipulate models.

The results have shown improvements of more than 90% in derivation times, and more flexibility thanks to the different types of bindings supported.

[S79] To close this gap, we present an approach for integrated variability management during software architecture design and implementation. The approach is an extension of LISA, a model and toolkit for architecture management and analysis. Variability modeling is provided by an additional view on a single consistent architecture model leading to a tight integration of variability and architecture modeling and implementation. Architects and developers are thus constantly aware of the variants they are working on and their implications on architecture design and implementation.

LISA (Language for Integrated Software Architecture), a model and toolkit for continuous architecture management and analysis of heterogeneous component-based software systems [7].

The LISA model is a meta-model for describing heterogeneous component-based and service-oriented software architectures [6,7]. The LISA toolkit is a set of plugins for the Eclipse IDE. It supports architecture modeling, visualization, and analysis of architecture models based on the LISA meta-model.

LISA currently provides technology bindings for J2EE, BPEL, Eclipse, Java, Spring, SCA, OSGi, and C#.

We use technology bindings (in this case OSGi [5] bindings) to bind architectural elements to implementation elements

Feature Modeling Plugin [11] is an Eclipse plugin for developing feature models. It integrates with the Rational Software Modeler or Rational Software Architect to support product line modeling. Features can be mapped to UML models during domain engineering to document how the variability is realized in design. During application engineering, the models are configured based on a valid feature selection. Similar to our approach, Feature Modeling Plugin supports linking variability models and (architecture) design models. However, the approach does not provide a single environment for architecture design and variability modeling. Developers need to switch between tools, which hinders a permanent view of the variability. Traceability of variability to implementation is not supported in the approach. In 5000 5002 general, the Feature Modeling Plugin approach is focused towards model configuration during application engineering.

Gears [12] is a development environment that supports variability modeling using so-called feature profiles where optional and varying features can be defined. Different kinds of assets (e.g. code files, requirements, documentation) can be linked to features and configured during application engineering. Several third party tools are integrated in Gears. The Rhapsody/Gears bridge supports the configuration of UML and SysML models based on feature choices in Gears feature profiles. The Gears approach is focused on configuration during application engineering. In contrast to our approach, Gears does not support variability modeling, architectural modeling, and implementation in a single, integrated environment. A permanent view on the variability is not provided because of the need to switch between tools.

Dopler [14] is an approach that supports decision and asset modeling to capture the variability of the system and its realization. Dopler supports the definition of new asset types to reflect the specifics of a particular product line. To link assets and decisions, inclusion conditions can be defined which have to be fulfilled for a particular asset to be included in a product. During product derivation, products can be automatically generated based on the inclusion conditions. The Dopler approach focuses on decision and asset modeling in a way that they can be efficiently used for product derivation. Support for a permanent view of the variability during the actual development of the product line architecture and implementation is not provided. Also, Dopler does not support synchronization of architectures and implementations.

Pure::variants [13] is a variability and variant management environment integrated into the Eclipse IDE. Feature models support modeling the commonality and variability of the product line. Family models support modeling the assets that describe the system in terms of architectural elements. The family model is divided into components (representing functional features of the solution) that consist of logical parts of the software (e.g. classes, functions, variables). Physical elements (files) can be assigned to logical elements. This is similar to our approach where implementation bindings are used to link architectures and implementations. Pure::variants integrates with the Enterprise Architect modeling environment. Features defined in pure::variants can be linked to SysML and UML models. Similar to Gears, developers have to switch between tools to document how the variability is realized in the models. Also, pure::variants does not support synchronization of architecture and implementation.

FeatureMapper [15] is an approach and toolkit integrated into the Eclipse IDE that supports feature modeling and linking feature models to arbitrary EMF models. The mapping can be performed manually or, similar to our approach, using modeling event recording. Different forms of highlighting are used to visualize the dependencies between feature models and EMF models. In contrast to our approach FeatureMapper does not support traceability of features to implementations and synchronization of architectures and implementations.

[S80] Web applications, as it is desirable to have access to the information and main features of these applications in mobile devices. In face of this problem, this paper discusses the motivation and presents the evolution from a SPL in the e-Gov Web (e-Gov Web SPL) domain to a SPL in the mobile domain (e-Gov Mobile SPL) having in mind the need to supply market demand. The conducted evolution was supported by the PLUS approach (Product Line UML-Based Software Engineering) and by the features model. Furthermore, this work debates the main results obtained through some e-Gov Mobile SPL instantiations in the precision livestock domain.

This paper presented the changes made in the e-Gov SPL to obtain an e-Gov Mobile SPL. In particular, a service bus that contemplates the communication between Web applications and their corresponding mobile applications was implemented in the Titan Framework (some of the existing components in its repository were refactored in order to be able to interact with the incorporated services); and, a code generator to create mobile applications (Titan Architect Mobile) was implemented.

The main motivation on conducting this work was the need to migrate several web systems, previously developed with the support of the e-Gov Web SPL, to a mobile platform, besides allowing the communication between the applications in both platforms (that is, Web and mobile).

To facilitate the e-Gov SPL Mobile instantiation, it was created a code generator in the fourth and last evolution cycle, **named Titan Architect Mobile**, which allows the application engineer to generate the mobile application code. For this, this generator scans the files in markup language (XML) from the Web application, responsible for the parametrization of the Titan Framework components, generating a Java code that composes the mobile application project. The basis of this project is also fixed and consists of a structure of directories according to the Android pattern and a collection of auxiliary classes, named helpers, which implement functionalities common to all members of the product line.

[S81] We use a pacemaker product line to evaluate different techniques for modeling crosscutting variabilities. These are real-time, embedded, and safety critical systems, that have been successfully developed in industry using MDD and software product line practices [26].

A pacemaker is an embedded medical device designed to monitor and regulate the beating of the heart when it is not beating at a normal rate

Based on our experiences using Rhapsody [13] as an OO modeling tool

The work described here provides a preliminary comparison of the OO and AO approaches in modeling crosscutting variabilities, based on experience with a product line case study.

Model Driven Development (MDD) [29], [30], [37] is a software development approach that uses diagrams to communicate and uses models to understand and validate the designs, as well as to help software implementation, deployment and maintenance. It often uses the Object-Oriented paradigm [28] to abstract the system functionality into models.

[S83] We extended a commercial software tool to support top-down as well as bottom-up traceability in product families, from the family feature map all the way down to implementation files. At the code level, both newly developed and commercial-off-the-shelf components are accommodated. The tool has been validated by (bottom-up) filling the tool’s reuse base with features, components, documentation files, etc. from six related products in the Next Generation Network service domain, and next deriving a seventh product from this reuse base.

Together® ControlCenter™, which provides extension mechanisms such as configuration files and integration of Java implementation modules. Starting with Together® ControlCenter™, we developed a tool that supports top-down as well as bottom-up traceability, from the product family feature graph all the way down to implementation files.

Traceability between Product CM and implementation. Elements in a Product CM are associated with their implementation (see the NavigationLink between Product CM and Implementation). The NavigationLink is provided by association Implements defined between each design decision and the implementation assets solving it (e.g. executables like COTS components, source code modules, and associated documentation files).

next generation networks (NGNs) service domain. NGNs integrate hybrid telecommunication networks (like fixed telephony, packet switched and wireless networks) via middleware platforms

[S84] In this paper, we present a novel approach to implementing product line architecture. It combines a code generation and separation pattern with an architecture-based code annotation technique. It can support product line variability in platforms, functions, and implementation mechanisms that are at different degrees of granularity. The goal is to maintain architecture-implementation conformance and increase code reusability.

Product line architecture (PLA) [6] is an important application of software architecture in development of a family of software products, or a software product line [22]. It captures architectural commonality and variability among products of the product line. Similar to a single system’s architecture [23, 24], PLA is commonly characterized as a configuration of components connected via explicitly-defined interfaces.

We have developed a toolset called xLineMapper to support the PLA implementation approach. It includes a template-based code generator and an annotation processor that can be used to develop and update the implementation of a PLA model. It also includes a code visualizer that can automatically highlight the code fragments of a PLA component that are related to different product line features.

An architecture-based code annotation is defined as a Java annotation (e.g. @Optional) wrapped by a Java block comment (i.e. /\*…\*/), and can be attached to any code element (e.g. variable, method, lines of code) as a regular Java comment does.

ArchJava [1] and Archface [25] are architecture implementation techniques both based on language design. They include new programming language elements to express architecture in source code.

First, ArchStudio is an **architecture development environment**. That is, the ArchStudio developers have identified recurring principal design concerns that occur in many domains and projects, and attempted to support these. ArchStudio has built-in support for modeling the hierarchical structure of complex systems, the types of various components, connectors, and interfaces, product-lines of systems that are related by a common base, and so on.

he elements drawn with solid lines are core elements. The elements drawn with dashed lines are variation points related to three product line features respectively:

[S85] This paper presents an experience on the implementation of SPL variabilities using AOSD. The Spring framework is used to configure the components of an existing SPL architecture. This paper compares the AOSD version of an SPL with an existing component-based one. Lessons learned are discussed regarding the use of AOSD in component-based SPL configured with the Spring framework

Reuse techniques such as Component-based Development (CBD)

The results achieved in relation to the implementations of existing variabilities using aspects suggested that, in the context of the LPS-BET, the two implementation alternatives can be considered roughly equivalent, with one exception: a solution based on components accurately reflects the architecture designed to the SPL. This is because in the version that uses AOSD, due to the Spring configuration, the original components are still connected to other components, although the operations are performed from the aspect, unlike the architecture designed for the LPS-BET, which may affect the future maintenance of the SPL

[S86] In this paper, we describe the limitations of the current practice of combining heterogeneous components in a product line and describe the challenges that arise from software supply chains. We introduce a model-driven approach for automating the integration between components that can generate a partially or fully configured variant, including glue between mismatched components. We analyze the consequences of using this approach in an industrial context, using a case study derived from an existing supply chain and describe the process and roles associated with this approach

When a system is formed from components that do not conform to a common architecture, these components are usually referred to **as heterogeneous components** [11]. When heterogeneous components have to be integrated, there might be mismatches between their interfaces, which have to be bridged by glue code. For instance, a set of interfaces might contain different numbers of methods (interface splitting), method parameters can be passed in different forms, e.g., as a struct vs. a list of separate parameters, methods having the same name might implement somewhat different functionality (functional splitting)

a distinction is made between variation points that relate to product features, denoted as functional variation points (FVP), and variation points that describe alternative suppliers, denoted as supplier variation points (SVP). In the ZigBee case, there are two supplier variation points - one for the MAC layer and one for the Network layer, and a list of functional variation points

IBM Rational Software Architect (RSA)

JET is typically used in the implementation of a "code generator". A code-generator is an important component of Model Driven Development (MDD).

[S88] . In this paper, we present SVL Tool, a plug-in for the Case Tool DB-Main which enables software engineers to model feature models, map them to database schema elements, and finally produce a new database schema including only the selected features. We present the Simple Variability Language, a language designed on the basis of the Common Variability Language. We also present our results of applying SVL Tool to a case study, an Electronic Medical Records software program widely used in Canadian primary health care.

SVL Tool, a plug-in for the DBMain CASE tool that supports the management of databaserelated variability.

The next step of this work would be to work directly with the SQL code without having to work with database schemas as a medium.