Service-Oriented Product Lines: Towards a Development Process and Feature Management Model for Web Services

Sebastian Günther
Very Large Business Applications Lab
School of Computer Science
Otto-von-Guericke-Universität Magdeburg
sebastian.guenther@ovgu.de

Thorsten Berger
Business Information Systems
Department of Computer Science
Universität Leipzig
berger@informatik.uni-leipzig.de

***Abstract*—Service-Oriented Architecture fosters the loose coupling of services aimed at maximizing flexibility, adaptability and configurability. Services of different providers can easily be integrated into a common framework with standardized technology like Web Services. A Software Product Line depicts a systematic software reuse approach by handling various types of flexible software artifacts that form a common platform and are the basis for deriving concrete products. This paper contributes towards the combination of both concepts by proposing a differentiated development process for Software Product Lines implementing a Service-Oriented Architecture. An extensive example shows how parts of this process can be solved technically with already developed methods for feature modeling and management using Web Services.**

I. Introduction

Software development has always been a rather sophisticated task. Requirements, functional and non-functional, described in natural language are transformed into source code through a number of steps. A vast number of artifacts is generated in the development process: architectural descriptions, software interfaces, source code, documentation and much more [11]. All those artifacts contain valuable information about the software. Much information is implemented and re-implemented into different systems. A Software Product Line (SPL) helps to structure the artifacts and to identify and manage commonalities and variability for a family of related software products.

In modern IT infrastructures with a multitude of protocols, languages and technologies, already existing applications need to be integrated thoroughly. Constant changes, ranging from business requirements to technological issues [5], demand high flexibility. SOA envisions a loosely-coupled net of business services, published, requested and invoke with standardized languages and protocols. Such services can be used and reused by different service consumers and quickly adapted to unanticipated changes.

The combination of these two concepts exposes a great potential to provide solutions for the current challenges in software development and infrastructure management. Some contributions in the first Service-Oriented Architectures and Software Product Lines Conference (SOAPL2007) considered the systematic comparison of the two concepts [12], proposed a concept to identify and specify reusable services [18] and suggested the definition of a Service-Oriented Product Line (SOPL) [28]. This position-paper contributes to the combination of both concepts by proposing a differentiated development process for Software Product Lines implementing a Service-Oriented Architecture. An extensive example shows how parts of this process can be solved technically with already developed methods for feature modeling and management using Web Services. For this, the authors do not claim the contribution, but instead follow the original contribution of [2] to reuse the ideas and example stemming from Kästner et al. as a base. In the following paragraphs, we first present background information to SPL, SOA and Web Services. We continue with a discussion of the development process for SPL and point out special considerations when developing a SOPL. This is followed by an example web store in which we sketch the application of an algebraic feature model and show how WSDL artifacts can be used to model features and refinements. A discussion of possible tool support and the related work precedes the conclusion.

II. Background

A. Software Product Lines

The concept of Software Product Lines (SPL) addresses the challenge of structuring and systematically reusing software development artifacts[1]. The goal of an SPL is to provide valuable production assets that help to implement concrete members of a software family. According to Withey, a "product line is a group of products sharing a common, managed set of features" [32]. Kästner et al. supplements: "Each feature represents an increment in functionality relevant to stakeholders" [15]. The core of SPL form features that represent functional and non-functional requirements. The variability management, meaning the modeling and implementation of commonalities and variability, is one of the central concept of SPL [30]. SPLs originate from the embedded systems domain. However, they are not only used for e.g. mobile phones [16], but also for complex financial software systems [30].

B. Service-Oriented Architectures

Service-Oriented Architectures (SOA) envisions a web of loosely-coupled, autonomic services that are published and queried with standardized languages to be used in flexible IT infrastructures. Typical properties of services in a SOA are, among others, self-containment, coarse-grained interfaces, reusability and composability [13]. The state-of-the-art for implementing a SOA can be considered in the Web Services technology [8] and the Enterprise Service Bus (ESB) concept [5]. Web services describe business services via standardized, XML-based interfaces - the Web Service Description Language (WSDL). These interfaces specify the operations and messages for communicating with other services [8]. Finally, the ESB is the central backbone of a SOA. It provides a uniform bus for information routing and the overall process flow. All types of applications, databases,

[1]Other concepts that address similar problems are Aspect-Oriented Programming [17], Feature-Oriented Programming [21], Generative Programming [6], Model-Driven Development [23] and Domain-Specific Modeling [16]

TABLE I
DOMAIN ENGINEERING DEVELOPMENT PROCESS (FROM [11])

Step	Sub step	Purpose	Artifacts
Analysis	Product line definition	Overall problem and context description	Text (informal)
	Problem domain scoping	Identification and selection of problems to be solved and harmonization with features	Problem description (informal), feature description (informal), domain model, feature model
	Solution domain scoping	Describes product line and product requirements, scoping the whole product line	Requirements specification, SPL and product description (informal)
Design	Product line architecture development	Specification how SPL produces concrete products and maps requirements to architecture	Architecture description, common design features descriptions, variable features description (variability points)
	Product development process	Overall processes to identify production assets and describe their use by roles and development context	Product development process description
Implementation	Implementation asset provisioning	Provide basic components for SPL via developing, reusing or buying	Components and accompanying documents (documentation, tests, tools)
	Process asset provisioning	Implements the product development process description with concrete documentation, guidelines and tools	Software development tools, documentation and guidelines (informal)

TABLE II
APPLICATION ENGINEERING DEVELOPMENT PROCESS (FROM [11])

Step	Purpose	Artifacts
Problem Analysis	Formulate the problem to be solved by the concrete member	Problem description (informal)
Product Specification	Specify the concrete member	Product requirements specification
Collateral Development	Produce additional not-executable artifacts	User documentation, packaging
Product Implementation	Specify concrete product design, develop executables and test cases, testing	Concrete components, test cases
Deployment	Deploy the concrete member	Complete member

legacy systems and ERP, can be connected to the ESB with a Web Service wrapper. With such a uniform interface definition, the vision of a completely integrated information system becomes true [5]. Recent examples for Service-Oriented Architectures are e.g. power management [19] or a generic market infrastructure [4]

C. Web Services

Web services are ”software applications that can be discovered, described and accessed based on XML and standard Web protocols” [7]. The core of a web service can be considered in its WSDL definition. This description is twofold. The *abstract definition* describes the service like a component, with its interface, operations and messages. A *concrete definition* describes concrete bindings of the operations to the so-called endpoints - they contain a data description, a physical address and the protocol information [8]. In an application environment, we distinguish between three roles: A service broker, provider and consumer. The provider registers a service at the broker, for example in a UDDI repository. The consumer discovers a service managed by the broker and calls the service at the provider. All parties are using SOAP, an XML-based protocol standard handling requests and responses of Web Services calls [13]. A good overview of other Web Service standards is given in [29]. Past and ongoing research targets fields like the efficient management of Web Services [9], semantic extensions of the WSDL [20] [1] and automated Web Service composition [22].

III. DEVELOPMENT PROCESS FOR SOFTWARE PRODUCT LINES

Existing development processes for SPLs are closely related to traditional software engineering methods. However, it differentiates between the two processes Domain Engineering (”develop for reuse”) and Application Engineering (”develop with reuse”). We assume that the reader is familiar with such processes and thus only point out some SPL-specific points in this section. We consider the method suggested in [11], which is further detailed in Table I for the Domain Engineering and in Table II for the Application Engineering.

The *Domain Enginnering* begins with the *analysis* phase. Its sub step *problem domain scoping* is an activity that stems from classical domain engineering methods [6]. It targets to capture the domain-specific concepts and their relationships to deepen the knowledge of developers and domain experts. A domain model represents this knowledge of the solution. In traditional software engineering, the concepts are mapped to databases or classes to represent a computational model of the domain. SPL uses in essence the same idea: Map domain concepts to software entities. Its distinguishing characteristic is the identification of variants as features. In essence, features state ”anything users or client programs might want to control about a concept” [6]. At this point, the *solution domain scoping* intersects with the previous phase. Not only the domain concepts, but also functional and non-functional properties of the SPL are considered, with some restrictions, as features. Functional requirements stem from the domain-model and additional functional specification. Certain non-functional requirements, like performance or security, can also be identified as features. Both are put in a central feature model that represents the complete software family. In order to effectively describe a concrete software, features are classified as mandatory, optional, variant or external [25], and can thus form rich relationships that describe software at a higher abstraction level. From this description, individual members of the family are specified by a selection of features, called a concrete configuration of the product line. Subsequently, the *design* phase now further details the SPL. Important considerations about the overall architecture and the software entities are made. Technological restrictions, like supported frameworks, libraries, buses and used programming languages are specified. Within the architecture, the common parts and variable parts are covered. The latter identify where variants of SPL members can be implemented. After these discussions, the development process for the SPL members is sketched. Lastly, the *implementation* provides core artifacts of the SPL by either buying or developing. For testing the feasibility of the SPL and to further detail the separation of variable and common parts, prototypes for some members can be developed and tested. If the technological base for the SPL is fully developed and functioning, the abstract development process for SPL members is completed with concrete guidelines and tools.

After the SPL has been designed with a complete coverage of the steps, the *Application Engineering* now constructs a concrete member of the SPL by following the steps described in Table II. Beginning with the overall problem the member tries to solve, a complete description of the requirements is developed. They are corresponding

to the already identified features in the *Domain Engineering*. Thus, the member implements a subset of all possible features. Next, the concrete member and its accompanying documentation are developed. In this phase, developers are mapping identified features to software entities. A mapping is typically not accomplished on a one-to-one basis from features to classes. Kästner et al. distinguishes between a *compositional approach* (features are implemented as feature modules) as well as an *annotative approach* (explicit and implicit annotations of the source code) [15]. It's essentially a question of feature granularity and scope which method can be used. Straight forward additions like a new product category in a web store may be implemented in a discrete feature, since other parts of the system access all items via a standardized interface. Logging mechanisms may require recording in distinguished steps of the application: At the presentation layer when the customer adds an item to the basket, and at the database level to log the result of an SQL expression. This requires fine-grained additions on the method level, either as direct calls to a logging method or even the explicit change of the source code receiving the results of the database transaction. A careful consideration of feature granularity and their implementation can populate a software repository with reusable artifacts to be used with other members as well. Once all executables are developed, they need to be tested and are finally packaged and deployed.

In summary, the whole process of identification, modeling and implementation of features, denoting functional and non-functional requirements of the combined solution and problem domain, and their management is the central issue of SPL engineering [30]. For a detailed discussion of variability and especially their realization, further information can be found in [25].

IV. Development Process for Service-Oriented Product Lines

An SPL can be implemented with different technologies. Choosing a Service-Oriented Architecture leads to more restrictions and other changes in the general development process for the Domain Engineering. The *analysis* takes only little impact from this decision. Its domain model and feature model are chosen technology-independent to be used with different complete architectures. However, if restrictions are known prior to this phase, then the concrete modeling language can be chained to the technology to use e.g. transformation tools that map models to software entities. In the case of SOA, models can be enhanced with additional properties that define the feature granularity. This allows a special generator to produce customized WSDL specification skeletons. Also, certain requirements may be either captured as features with services or as part of the overall architecture, especially the ESB.

This leads directly to the *design* phase. A SOA consist of the central ESB and connected web services [5], meaning architecture choices for an SOPL are limited to these paradigms. The ESB is responsible for the connection of Web Services, routing messages between different services, lifecycle and connection management as well as numerous QoS and security issues [5]. Its functionality is the common part of the SOPL. The ESB can also be responsible for a large number of requirements or legal aspects such as compliance. A careful choice can lead to a better focus of providing services to the bus and leave other aspects to a developed technology. Considering Web Services, many options exist for the concrete realization: Granularity (components, classes, methods), implementation (compiled and interpreted code) and software entities (applications, databases, systems). A service itself is differentiated into an *interface* and its textitimplementation. These options must be selected appropriately. Restrictions can be made not only to the WSDL describing the service interface, but also the message exchange protocol: A unique version can be chosen to ensure compatibility between Web Services and the ESB. Another important point is to regard variability management in Web Service descriptions. All these considerations form the base for the *product development process*, which specifies overall requirements for implementing different services.

The last phase *implementation* begin with providing the core assets. With a thorough requirements specification, a suitable ESB can be either selected or developed. Basic services can be provided with different ways, according to the required granularity. Service implementation can encompass components or systems, provided by in-house developments or bought from the market. The other way is to use existing services that are offered in repositories via UDDI [2]. Once the major parts are acquired, it remains to detail the implementation process for the services. Considering the availability of service descriptions, two different processes are discovered. A *full SOPL process* is designed at developing interfaces and its implementations in concert: Auxiliary guidelines for implementation technology are needed. This process can also incorporate different parties, like service providers, in the overall SOPL member implementation process. When no access to the implementation is required or available, the *light SOPL process* is sufficient. The consumer only considers interfaces and decides how different features and variants are represented. He specifies the interfaces and uses external resources to provide the implementation.

Discussing the overall advantages, an SOPL can profit from a careful choice of the ESB. It can be responsible for a large part of the requirements and since it is a purchasable asset may actually speed the development process, especially if the same ESB is reused in other SPL. The other part is the universal view on services and features via a WSDL. The interface description hides implementation details; feature granularity is embedded into the concrete realization. For illustrating these steps and to show the practical realization of this process, the next chapter shows selected steps with an example.

V. Example for a Service-Oriented Product Line: Web store

In this section, we show how already developed methods can be used to practical execute parts of the SOPL process. We regard a case of *Domain Engineering* a web store. The three following sections show how an algebraic feature model for the web store looks like (*analysis*), details the use of the feature model to show the variability management *analysis*, and finally presents a method of variability management with WSDL*design/implementation*. As a base, we take with permission the example from [2]: The therein proposed formulation of an algebraic feature model for the web store is extended with additional detail, and supplemental we cover the variability management for WSDL with the AHEAD tool suite [26] in greater extend.

In the web store, customers are browsing a catalog of products, log into the store and place an order. When processing orders, the credit worthiness is rated and, if accepted, the order is shipped to the customer. All these requirements are implemented as Web Services. Figure 1 shows the basic structure and workflow.

A. The Feature Model

After the initial setup of the SOPL, we now engage *problem domain scoping* and want to build a corresponding feature model.

[2]Although a shared repository from IBM, Microsoft and SAP has been closed in January 2006 [13], the general concept remains useful

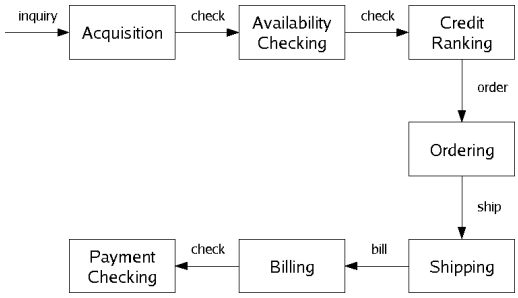

Fig. 1. Basic services of the web store (from [2])

We use an algebraic notation for the model [3] [2]. The root of the model is the concept Store, denoted by STORE. It consists of the mandatory features ACQ, CHK, CRD, ORD, SHP, BIL, and PAY. The names are corresponding to Figure 1 (starting from Acquisition to Payment Checking) since each feature is implemented as a service. The following equation states that concept BASE consists of several other concepts or features:

$$Base = \{Acq, Chk, Crd, Ord, Shp, Bil, Pay\} \quad (1)$$

We can now further distinguish the features. CRD can be realized with either an independent agency AGC or via the self-explanation of the bank (BAK). SHP can be done via surface transports (SUR), which includes standard (STD) or express (EXP) mail, or via airmail (AIR). The corresponding equations are:

$$Crd = \{Agc, Bak\} \quad (2)$$
$$Shp = \{Sur, Air\} \quad (3)$$
$$Sur = \{Std, Exp\} \quad (4)$$

The realization of a program is expressed by a composition ("•") of its implemented features. The following equation expresses the concrete member of the BASE SOPL, named STORE1, that implements all possible features as specified before.

$$Store1 = Acq \bullet Chk \bullet Agc \bullet Bak \bullet Ord \bullet Std \quad (5)$$
$$\bullet Exp \bullet Air \bullet Bil \bullet Pay \quad (6)$$

B. Variability Management with the Feature Model

After having used the web store for some time, customers make suggestions and demand new features. The first feature is discounting (DISC). When ordering a bigger quantity of items, users get a discount on the total price. Another feature is the order traceability (TRCE). Users want to check where their parcel is physically located. This feature requires to fetch data from an external web service provided by the postal service and to show the location with another external map application. These features touch existing services and add new services to the SOA. Figure 2 shows which services are affected when adding DISC. The changes address CHK (discount may lead to higher total price - can the customer afford this?), BIL (reduced price imposes a certain quantity and a special customer status) and PAY (again, reduced price must be charged).

The discounting concept thus *refines* four basic features. Refinements are expressed with a "Δ" suffix.

$$Disc = \{\Delta Crd = \{\Delta Agc, \Delta Bak\}, \Delta Bil, \Delta Pay\} \quad (7)$$

We now build a new web store STORE2 with the complete DISC feature and the BASE, but limit the shipment to STD.

$$Store2 = \{Base - \{Exp, Air\}\} \bullet Disc \quad (8)$$
$$Store2 = Acq \bullet Chk \bullet Agc \bullet Bak \bullet Ord \bullet Std \quad (9)$$
$$\bullet Bil \bullet Pay \bullet Disc \quad (10)$$
$$Store2 = Acq \bullet Chk \bullet Agc \bullet Bak \bullet Ord \bullet Std \quad (11)$$
$$\bullet Bil \bullet Pay \bullet \Delta Agc \bullet \Delta Bak \bullet \Delta Bil \bullet \Delta Pay \quad (12)$$

Assuming that the combination of a BASE and a DISC feature yields a combined feature (e.g. $Bil' = Bil \bullet \Delta Bil$), the final representation of STORE2 is

$$Store2 = Acq \bullet Chk \bullet Agc' \bullet Bak' \bullet Ord \bullet Std \quad (13)$$
$$\bullet Bil' \bullet Pay' \quad (14)$$

C. Variability Management with WSDL

Inside an SOPL, features and their variants are directly implemented as Web Services, specified by an WSDL. This mapping is intuitive: The high-level view off an service description is an interface to an arbitrary software entity, like components or whole systems. Questions of feature granularity are addressed via the universal representation of WSDL. The first step is to develop a basic WSDL description for the features. Due to space limitations, we focus on the data types of feature BIL and alternations when adding DISC. We omitted abstract definition of messages and the interface, and concrete definition for a binding and the service. Figure 3 A) lists the relevant WSDL data type specification for BIL.

To obtain variation in the feature model, we introduced refinements of basic features. The refinements just contain enough information that the composition of the basic and the refined feature yields the composed variant. For the WSDL, we need a way to specify refinements that add or change elements at certain positions in the document. A straight forward solution could combine an expression for the position of the refinement and an XML fragment to be inserted or update at this position. This idea is shown in [26] with the AHEAD tool suite and its component XAK for refining XML artifacts. An example for this approach is given in Figure 3. In B) we see the augmented part of the WSDL specifying the available types. Two identifiers are added: `xak:artifact="STOREbillOutput"` for the overall element, and `xak:module="billOutput"` for the sequence in which all single elements are contained. In C) we see an XAK refinement. DISC adds the elements `discount` and `discountedPrice`. The refinement begins with a declaration of the artifact that is to be refined. The next element `xak:extends` is an operation to add code to the module identified as `billOutput`.

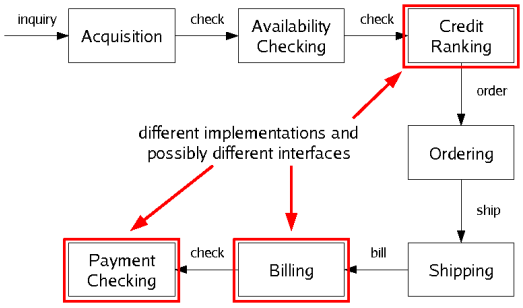

Fig. 2. Affected services of the web store when adding DISC (from [2])

```
<types>
 <xsd:schema
      targetNamespace="http://www.example.com/bill.xsd"
       xmlns="http://www.w3.org/2000/10/XMLSchema">
  <xsd:element name="Item">
   <xsd:complexType>
    <xsd:sequence>
     <xsd:element name="itemName" type="xsd:string"/>
     <xsd:element name="itemID" type="xsd:long"/>
     <xsd:element name="quantity" type="xsd:integer"/>
    </xsd:sequence>
   </xsd:complexType>
  </xsd:element>
  <xsd:element name="CalcBillInput">
  <xsd:complexType>
    <xsd:sequence>
     <xsd:element name="cutomerID" type="xsd:long"/>
     <xsd:element name="orderID" type="xsd:long"/>
     <xsd:element name="items" type="ItemOrder"
          minOccurs="1" maxOccurs="unbound"/>
    </xsd:all>
   </xsd:complexType>
  </element>
  <element name="CalcBillOutput">
   <xsd:complexType>
    <xsd:sequence>
     <xsd:element name="cutomerName"
          type="xsd:string"/>
      <xsd:element name="customerAddress"
           type="xsd:string"/>
     <xsd:element name="items" type="ItemOrder"
          minOcurs="1" maxOccurs="unbound"/>
      <xsd:element name="totalPrice"
           type="xsd:integer"/>
     </xsd:all>
    </xsd:complexType>
   </element>
  </schema>
</types>
```

A) Schema definition for ItemOrder, CalcBillInput und CalcBillOutput

```
<!-- Other definitions omitted --!>
<element name="CalcBillOutput"
     xak:artifact="STOREbillOutput">
 <xsd:complexType>
  <xsd:sequence xak:module="billOutput">
   <xsd:element name="cutomerName" type="xsd:string"/>
   <xsd:element name="customerAddress"
        type="xsd:string"/>
   <xsd:element name="items" type="Item" minOcurs="1"
        maxOccurs="unbound"/>
   <xsd:element name="totalPrice" type="xsd:integer"/>
  </xsd:all>
 </xsd:complexType>
</element>
<!-- Other definitions omitted --!>
```

B) Schema definition for CalcBillOutput with XAK extensions

```
<xak:refines xak:artifact="STOREbillOutput">
 <xak:extends xak:module="billOutput">
  <xak:super xak:module="billOutput"/>
  <xsd:element name="discount" type="xsd:integer"/>
  <xsd:element name="discountedPrice"
       type="xsd:integer"/>
 </xak:extends>
</xak:refines>
```

C) Refinements for billOutput to add elements “discount” and “discountedPrice”

```
<!-- Other definitions omitted --!>
<element name="CalcBillOutput"
     xak:artifact="STOREbillOutput">
 <xsd:complexType>
  <xsd:sequence xak:module="billOutput">
   <xsd:element name="cutomerName" type="xsd:string"/>
   <xsd:element name="customerAddress"
        type="xsd:string"/>
   <xsd:element name="items" type="Item" minOcurs="1"
        maxOccurs="unbound"/>
   <xsd:element name="totalPrice" type="xsd:integer"/>
   <xsd:element name="discount" type="xsd:integer"/>
   <xsd:element name="discountedPrice"
        type="xsd:integer"/>
  </xsd:all>
 </xsd:complexType>
</element>
<!-- Other definitions omitted --!>
```

D) Composition of the base and its refinement

Fig. 3. Complete WSDL specification of an basic WSDL data typ description and its XAK variants

The `xak:super` statement marks the exact position where the following code in line 4 and 5 is going to be inserted. When the refinement and the base are composed (in feature model notation: $Bil' = Bil \bullet \Delta Bil$) we yield the result in D): The additional statements are merged into the final representation.

VI. Discussion and Related Work

The example showed that current feature modeling methods can be applied to services as well - especially if only their interfaces are considered. With the usage of existing tools, we can also facilitate variability management with WSDL descriptions. The AHEAD tool suite already implements refinements for XML artifacts. The identified phases in the SPL development process that are especially treated in SOPL development (*problem domain scoping*, *product line architecture development* and *implementation asset provisioning*) can be managed with the proposed methods as well. Open issues are a detailed description of the processes and possible differences when using a *full SOPL process*.

Although a valid representation, the formalism of a mathematical feature notation and the specifics of WSDL are not suitable for the end-user. The most promising method to conquer the abstraction-gap in current software modeling is the use of Domain-Specific Modeling for full code generation [16]. In this approach, the infrastructure engineer, language engineer and method user [14] discuss and specify the problem. Together they generate a domain model and subsequently a domain-specific language (DSL) [31] that is suitable to bridge the gap. With this method, users apply modeling concepts of the DSL to specify the solution they want. The DSL then translates specifications to full code, which gives them the character of a configuration language. With a mapping from DSL to the feature model, users can specify the concrete member, and the DSL interpretation configures, implements and deploys the member of a SOPL. In a variation for web portlets, this is proposed in [27]. This approach also has interesting points into the domain model view on SOPL: Can parts of the model be implemented with different services, and can we thus implement the whole product line with a high-level view of integrated models? If different services are also instances of another SOPL, an implementation vehicle for Very Large Business Applications [10] could be developed.

Related work already showed how the algebraic feature model can be utilized with SOPL and also gave a clear discussion of benefits when SOA is approached with a feature perspective [2].

Other considerations that could not be covered here are the discussion of service classification into molecular and orchestrating services [18], a complete architectural modeling for SOA [24] or discussion of different domain-analysis techniques like DEMRAL, FeatuRSEB and more [6].

VII. Conclusion

This paper gave an overview to Software Product Lines, Service Oriented Architectures and Web Services. It showed differences between a traditional software development process and the development of an SPL. We then further differentiated the aspects of developing a Service-Oriented Product Line. In the main part of the paper, we gave an extensive example of a web store. We showed how the service can be modeled leveraging an algebraic notation and, furthermore, how feature additions and refinements can be handled. Then we continued with implementation details using WSDL. We presented a solution to manage the variability of service descriptions by customizing the existing tool suite AHEAD. As pointed out in the discussion section, we believe that a further integration of Feature-Oriented Programming and Domain-Specific Modeling using an Domain-Specific Language is an ideal base for a tool supporting the development of Service-Oriented Product Lines.

Acknowledgements

We thank Christian Kästner for his insightful comments and the permission to use the example in [2].

References

[1] A. Ankolekar, M. Burstein, J. R. Hobbs, O. Lassila, D. L. Martin, S. Narayanan, M. Paolucci, T. Payne, K. Sycara *et al.*, "Daml-s: Semantic markup for web services." The Emerging Semantic Web: Selected Papers from the First Semantic Web Working Symposium, 2002, pp. 131–152.

[2] S. Apel, C. Kästner, and C. Lengauer, "Research challenges in the tension between features and services," Proceedings ICSE Workshop on Systems Development in SOA Environments (SDSOA). New York, NY, USA: ACM, 2008, pp. 53–58.

[3] D. Batory, J. N. Sarvela, and A. Rauschmayer, "Scaling step-wise refinement," *IEEE Transactions on Software Engineering*, vol. 30, no. 6, pp. 355–371, 2004.

[4] I. Chao, R. Brunner, F. Freitag, L. Navarro, P. Chacin, O. Ardaiz, and L. Joita, "A decentralized grid market infrastructure for service oriented grids," *Wirtschaftsinformatik*, vol. 50, no. 2, pp. 25–30, 2008.

[5] D. A. Chappell, *Enterprise Service Bus: Theory in Practice*. Sebastopol, California, USA: O'Reilly Media, 2004.

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

[7] M. C. Daconta, L. J. Obrst, and K. T. Smith, *The Semantic Web: A Guide to the Future of XML, Web Services, and Knowledge Management: A Guide to the Future of XML, Web Services and Knowledge Management*. Indianapolis, Indiana, USA: Wiley Publishing, Inc., 2003.

[8] T. Erl, *Service-Oriented Architecture: A Field Guide to Integrating XML and Web Services*. Upper Saddle River, New Jersey: Pearson Education, Inc., 2004.

[9] D. Fensel and C. Bussler, "The web service modeling framework wsmf," *Electronic Commerce Research and Applications*, vol. 1, pp. 113–137, 2002.

[10] B. Grabski, S. Günther, S. Herden, L. Krüger, C. Rautenstrauch, and A. Zwanziger, "Very Large Business Applications," *Informatik-Spektrum*, vol. 30, no. 4, pp. 259–263, 2007.

[11] J. Greenfield, K. Short, S. Cook, and S. Kent, *Software Factories: Assembling Applications with Patterns, Models, Frameworks and Tools*. Crosspoint Boulevard, Indianapolis: Wiley Publishing, Inc., 2004.

[12] A. Helferich, G. Herzwurm, and S. Jesse, "Software product lines and service-oriented architecture: A systematic comparison of two concepts," *Proceedings of the First Workshop on Service-Oriented Architectures and Software Product Lines (SOAPL), Kyoto, Japan*, 2007.

[13] N. M. Josuttis, *SOA in Practice: The Art of Distributed System Design*. Sebastopol, California, USA: O'Reilly Media, Inc., 2007.

[14] D. Karagiannis and H. Kühn, *Metamodelling Platforms*, ser. Lecture Notes in Computer Science. Berlin, Heidelberg, Germany: Springer-Verlag, 2002, vol. 2455, pp. 451–464.

[15] C. Kästner, S. Apel, and M. Kuhlemann, "Granularity in software product lines," Proceedings of the 30th International Conference on Software Engineering (ICSE). New York, NY, USA: ACM, 2008, pp. 311–320.

[16] S. Kelly and J.-P. Tolvanen, *Domain-Specific Modeling: Enabling Full Code Generation*. Hoboken, New Jersey, USA: John Wiley & Sons, Inc., 2008.

[17] G. Kiczales, J. Lamping, A. Menhdhekar, C. Maeda, C. Lopes, J.-M. Loingtier, and J. Irwin, *Aspect-Oriented Programming*. Berlin, Heidelberg, Germany, New York, USA: Springer-Verlag, 1997, vol. 1241, pp. 220–242.

[18] J. Lee, D. Muhtig, M. Naab, M. Kim, and S. Park, "Identifying and specifying reusable services of service centric systems through product line technology," *Proceedings of the First Workshop on Service-Oriented Architectures and Software Product Lines (SOAPL), Kyoto, Japan*, 2007.

[19] T. Luhmann, J. Meister, and C. Wulff, "Serviceorientierte Produktplattform für das Energiemanagementsystem der Zukunft," *Wirtschaftsinformatik*, vol. 49, no. 5, pp. 343–351, 2007.

[20] S. Narayanan and S. A. McIlraith, "Simulation, verification and automated composition of web services," Proceedings of the 11th international conference on World Wide Web (WWW). Honolulu, Hawaii, USA: ACM, 2002, pp. 77–88.

[21] C. Prehofer, "Feature-oriented programming: A fresh look at objects," *Lecture Notes in Computer Science*, vol. 1241, pp. 419–443, 1997.

[22] J. Rao and X. Su, *A Survey of Automated Web Service Composition Methods*, ser. Lecture Notes in Computer Science. Berlin, Heidelberg, Germany: Springer-Verlag, 2005, vol. 3387, pp. 43–54.

[23] T. Stahl and M. Völter, *Modellgetriebene Softwareentwicklung. Techniken, Engineering, Management*. Heidelberg: Dpunkt Verlag, 2005.

[24] Z. Stojanovic, A. Dahanayake, and H. Sol, "Modeling and design of service-oriented architecture," vol. 5. IEEE International Conference on Systems, Man and Cybernetics (SMC), 2004, pp. 4147–4152.

[25] M. Svahnberg, J. van Gurp, and J. Bosch, "A taxonomy of variability realization techniques: Research articles," *Software Practice and Experience*, vol. 35, no. 8, pp. 705–754, 2005.

[26] S. Trujillo, D. Batory, and O. Diaz, "Feature refactoring a multi-representation program into a product line," Proceedings of the 5th international conference on Generative programming and component engineering (GPCE). Portland, Oregon, USA: ACM, 2006, pp. 191–200.

[27] ——, "Feature oriented model driven development: A case study for portlets," in *Proceedings of the 29th international conference on Software Engineering (ICSE)*. IEEE Computer Society, 2007, pp. 44–53.

[28] S. Trujillo, C. Kästner, and S. Apel, "Product lines that supply other product lines: A service-oriented approach," *Proceedings of the First Workshop on Service-Oriented Architectures and Software Product Lines (SOAPL), Kyoto, Japan*, 2007.

[29] A. Tsalgatidou and T. Pilioura, "An overview of standards and related technology in web services," *Distributed and Parallel Databases*, vol. 12, no. 2-3, pp. 135–162, 2002.

[30] F. J. van der Linden, K. Schmid, and E. Rommes, *Software Product Lines in Action: The Best Industrial Practice in Product Line Engineering*. Berlin: Springer-Verlag, 2007.

[31] A. van Deursen, P. Klint, and J. Visser, "Domain-specific languages: An annotated bibliography," *ACM SIGPLAN Notices*, vol. 35, no. 6, pp. 26–36, 2000.

[32] J. Withey, "Investment analysis of software assets for product lines," *Software Engineering Institute, Carnegie Mellon University, Technical Report CMU/SEI-96-TR-10*, 1996.