Software Product Line Engineering to Develop Variant-rich Web Services

Bardia Mohabbati, Mohsen Asadi, Dragan Gašević, and Jaejoon Lee

Abstract Service-Oriented Architecture (SOA) enables enterprise for distributed and flexible software development. SOA aims at promoting effective software asset reuse by means of encapsulating functionalities as reusable services accessible through well-defined interfaces. However, one of the challenging problems for the realization of this regard is an ever-increasing need for the design and management of variants of SOA-based solutions which require customization to meet stakeholders' individual functional and non-functional requirements. In this chapter, we have introduced 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.

1 Introduction

Nowadays enterprises and companies deal with several challenges for developing SOA-based solutions. To stay relevant with the global competition, they need to rapidly and cost-effectively develop and deploy stockholder-tailored services to meet a wide variety of their particular domain or targeted market sectors within a particular domain or targeted market sectors. These challenges motivate enterprises to shift from mass software production to mass software customization. A

Bardia Mohabbati
Simon Fraser University, Canada, e-mail: mohabbati@sfu.ca

Mohsen Asadi
Simon Fraser University, Canada, e-mail: masadi@sfu.ca

Dragan Gašević
Simon Fraser University, Canada, e-mail: dgasevic@acm.org

Jaejoon Lee
Lancaster University, United Kingdom, e-mail: j.lee@comp.lancs.ac.uk

trend towards developing software applications composed of *reusable software assets* for different requirement sets To enable mass customization in the context of Service-Oriented Architectures (SOAs), innovative software engineering methods and models need need: 1) to capture the knowledge of variable requirements and reflect variability of services 2) to support the reuse of services and all other software development assets 3) to enable the customizing and managing of service according to stockholders' functional and non-functional requirement [13, 35, 2].

Software Product Line Engineering (SPLE) is one of the most promising and well-established paradigms focusing on the development of software product lines [47, 12] based on the principles of variability modeling and mass-customization. SPLE research has proposed numerous approaches and techniques for the efficient production of similar software systems (also known as software families). Hence, the adaptation of SPLE approaches for mass customization is the center of attention now and has already been applied successfully to many enterprises [38]. Employing SPLE techniques results in the reduction of cost, effort, time-to-market and the improvement of quality, together decreasing the complexity of design, facilitating and expediting the customization, maintenance, and evolution of software. [47, 43, 12].

SPLE offers promising prospects to provide scalable solutions to the current challenges of the development, management and customization of Web services and generally SOA-based systems [13, 35, 34, 14] to which we refer, in the course of this paper, as *Service-Oriented Product Lines (SOSPLs)*. In this chapter, we provide a comparison of SPL and SOA from different perspectives then present a method for a systematic development of a family of SOA-based applications (i.e., SOSPL). The underlying idea of which is to guide the development process of an SOSPL and which extends the conventional SPLE life-cycle to support modeling, developing and managing variant-rich service-oriented applications.

This chapter is organized as follows: Section 2 introduces the basic concepts of SPLE and outlines some of the main SPLE activities. Section 3 presents a holistic comparison of SPL and SOA which focuses on reuse, architectural and variability aspects of the two paradigms. Section 4 introduces the end-to-end methodology for SOSPL development by focusing on the main engineering activities of the approach. Section 5 discusses in details the proposed approach, and conclusion is given in Section 6.

2 Software Product Line Engineering (SPLE)

SPLE addresses the issues of software reuse and mass-customization. An SPL or a software product family is defined as: *"a set of software-intensive systems, sharing a common, managed set of features that satisfy the specific needs of a particular market segment or mission and are developed from a common set of core assets in a prescribed way "* [12]. The 'particular market segment' refers to a domain (i.e., a business area) and the business strategies of an enterprise or organization whose

objectives of the business area are determined with changes in its stockholders' requirements in mind.

A key idea in SPLE is to capture the essential concepts of 'commonality' and 'variability' among a set of similar software products belonging to the same domain. Therefore, rather than describing a single software system, the model of software product lines describes the set of products in the same domain. A product line includes predicted variations that are introduced by tailoring the core assets using variation mechanisms. Variability introduced in SPLE is an abstraction that enables and facilitates customization. It empowers product derivation of different applications by explicit modeling and management of variation points which define decision points determining how the product family members may differ from each other [47, 56]. Variations along with their possible choices, functions or qualities, can be defined at each level of abstraction (e.g. requirements, architecture, or components).

SPLE relies on a fundamental distinction of *development for reuse* and *development with reuse* with aims at maximizing reusability and eliminating wasteful generic development of components used only once. This insight can be leveraged to improve software development life-cycle that SPLE shifts from the development of a specific application or individual system to a domain, in turn, leads to two characterized development processes commonly referred to as *domain engineering* and *application engineering* [47, 12].

SPLE shifts from the development of a specific application or individual system to a domain. This, in turn, leads to two characterized development processes which are commonly referred to as *domain engineering* and *application engineering*. Domain engineering models variability among product family members and develops the reusable software platform by focusing on *developing-for-reuse*. The software platform encompasses all software development artifacts that are liable to be reused. On the other hand, application engineering adopts the *developing-with-reuse* approach, where products are customized and derived from product family and reference platform which is constructed and developed in the domain engineering phase. Reuse of the software platform and binding variability for different applications are then enacted in application engineering. Differentiating these two development lifecycles allows for establishing the software platform, application customization, and product derivation.

Approaches to the analysis and construction of SPLs can be classified into three strategies: i) proactive, ii) reactive, and iii) extractive [22]. A proactive strategy is similar to the waterfall approach in conventional software engineering, where all product variations on the foreseeable horizon are analyzed and designed, while architectures for the target domain are defined and implemented upfront. This approach is appropriate for enterprises to foresee and plan ahead of their product line requirements well and that have available resource and time for a long development cycle. A reactive strategy is an incremental approach where only the product-line reusable assets needed in immediate terms are developed and built. Hence, this approach typically requires less upfront efforts than proactive. In a reactive strategy, one or several variations of software products can be analyzed, designed and im-

plemented in each development spiral. Such an approach is appropriate where the upfront requirements for product variations cannot be predicted well in advance or where enterprises have to to meet tight schedule, which is usually limited in resources, through the transition to an SPLE approach. An extractive strategy is between proactive and reactive ones and reuses existing software products as the product line initial baseline.

3 Comparison of SPL and SOA

SPL and conventional SOA-based approaches to software development share common goals with both promoting the concepts of reuse and foster organizations to recycle existing assets and capabilities rather than repeatedly redeveloping them for new software systems. Recent years have witnessed growth of research in the exploration of synergies of the combination of SPL and SOA. [13, 23, 35, 14, 26, 9, 59]. Even though two paradigms support software reuse, there are different perspectives and outlooks [36]. In this section, commonalities and differences corresponding to the two paradigms are discussed helping to enlighten how SPLE can be adopted and leveraged for the development and customization of a family of SOA-based applications. To compare SPL and SOA, we consider four main aspects including development processes, reusability notions, architectural styles, and variability modeling and management.

3.1 Development Processes

SPL and SOA follow different engineering goals. Thus, the activities associated with their software development life-cycles are different. One of the main objectives of SPLs is to reduce the overall engineering efforts required to produce a set of similar software applications by capitalizing on the commonality and by managing the knowledge of variability and customization. Therefore, the engineering goal of SPL is remarked as the systematic development and management of core assets and software platform in order to achieve the high level of reusability [22, 47, 43, 12]. In contrast, service-oriented approaches set the goal of achieving system agility and of enabling automation to cope with integration, inteoperability and dynamic execution in heterogeneous environments, and providing runtime flexibility [6, 20, 46]. Table 1 shows a summary of major life-cycle phases of two paradigms essentially including requirement and domain analysis, design and implementation and deployment.

- **Requirement and Domain Analysis:** Service-oriented design and development are basically based on an iterative and incremental process. The process is initiated with planning proportional to the requirements which, for a new application, are investigated in analysis phase. This process comprises of reviewing business goals and objectives that derive the modeling and development of business pro-

Table 1 Comparison of the major engineering activities of software product line engineering and service orientation

Engineering Paradigm		Requirement Analysis	Design and Implementation	Deployment and Maintenance	Main Engineering goals
Service-Oriented Engineering		• Planning and requirement analysis • Business process models • Service identification	• Business process specifications • Service construction	• Service publishing • Service matching • Execution and monitoring	o Integration and Interoperability o System agility through run-time flexibility o Dynamic execution
Software Product Line Engineering	**Domain Engineering** *Development for reuse*	• Product line scoping • Product line requirement analysis • Variability analysis	• Domain design • Domain realization • Domain testing	• Product line maintenance evolution	o Variability modeling o Variability management o Systematic reuse of assets for development of a software product family o Mass customization
	Application Engineering *Development with reuse*	• Application requirement analysis	• Application design • Application realization • Application Testing	• Application deployment	

cesses. Business processes and services are identified in a stepwise manner in the analysis phase the main objective of which is to facilitate [20, 45]. The main objective of the analysis is to facilitate the reuse (or reproposing) of the business process functionality through the identification and orchestration of services when constructing new applications.

The requirement-analysis phase in SPLE also consists of determining the requirement and using domain information. Nevertheless, SPLE focuses on the analysis and specification of requisites for the entire product family (i.e., product line). To this end, domain engineering of SPLE mainly concentrates on a systematic analysis and the settlement of variability of both functional and non-functional (quality) requirements performed by scoping the product line, analysing product line requirements, and identifying commonalities and variabilities among product line members. Requirement analysis in the application engineering life-cycle further focuses on the analysis and determination of prerequisites of individual stakeholders. In the application engineering life-cycle, requirement analysis is established for configuring reusable software assets developed and produced in the domain engineering life-cycle.

- **Design and Implementation:** Service-oriented design and implementation is followed by the design and specification of business processes and service components corresponding to the requirements. Service implementation and testing involves the discovery of available services in local or remote repositories and the development of services, using the specifications in the design phase.

  In SPL, domain design and implementation involve the detailed design and realizing the reusable software components for the entire product family. It starts with the domain design sub-process which consists of 1) defining and modeling the commonality and variability based on the domain-specific requirements identified in the requirement-engineering phase; 2) specifying the product-family reference architecture which provides a common, high-level structure for all the product-line applications. Furthermore, the domain design incorporates configuration mechanisms into the reference architecture for supporting variability management in order to enable further product

customization and derivation. The domain realization sub-process focuses on the implementation and the testing of each component planned and designed for the reuse in different contexts (i.e., the applications of the product line). The application design sub-process in the application design sub-process in the life-cycle of application engineering incorporated application specific adaptation and employs the reference architecture to refine and instantiate the application architecture. Afterward, the application realization sub-process focuses on the selection and configuration of reusable software components and testing for specific application- already contained product line architecture developed in domain engineering phase.

- **Deployment and Maintenance:** Service-oriented development deals with packaging, provisioning, publishing services, service-matching based on requirements of stakeholders, executing stakeholders-acceptance testing, and monitoring performance in the production environment. the SPLE development phase including the configuration and deployment of a final product is associated with application engineering with activities for custom-building systems according to the result of domain engineering.

3.2 Reuse in SPL versus SOA

Software reuse, as one of the important goals in software engineering, can improve the quality and productivity of software development. For this purpose, several software reuse approaches have been devised. Component-based software engineering (CBSE) facilitates software reuse and promotes quality and productivity.

CBSE focuses on the interoperability, reusability, and extensibility to facilitate fast delivery of scalable, evolving software systems with research on SOA being a modern instance of this vision, leveraging a logical framework by decoupling several logical units of functionality (services) [32]. This logical framework yields facility reusability through obviating the recreation of common services, thus the achievement of business goals by way of loosely connected services with their variability guided by SOA policies [6, 44].

Reuse in SPL vs. services in SOA have different characteristics (cf. Table 2). As mentioned, reusable assets in SPL encompass all the reusable software artifacts. A core asset is the most essential element of SPL since it is a common asset which is reused within multiple products and the reusability of which will largely determine the success of the whole product line [47]. For instance, the most distinguishable reusable assets in SPL context are as follows [12]:

- *Analysis and design models*: including the requirements and variability models, which describe the common and variable features for all family members
- *Domain models*: describing and representing all the entities and concepts that can be utilized in the context of software product families

Table 2 Reuse in SPL and SOA

Reuse Characteristic	Reuse in SPL	Reuse in SOA
Reuse Units	Analysis and Design Models (Requirement Models), Domain Models, Architectures, Decision Models, Software Components, Composition Models, Interfaces, Test Cases, Documentations	Service, Business Processes or Collaboration Templates, Application templates, Data Schema and Data Provenance, Policies and Business Rules, Test Scripts, Interfaces
Reuse Context	Software Family Members	Various Contexts
Coupling With Reuse	Tightly Coupled	Loosely Coupled
Reuse Method	Instantiation	Service Invocation Composition

- *Architectures*: specifying and determining which of the reusable components are needed for configuring executable applications and how to configure software families that best satisfy non-functional requirements
- *Design decision models*: specifying the family configuration model and determining how to derive software products based on specific requirements
- *Software components*: supporting variation points and implementing the required functionalities of software families
- *Interfaces*: enabling different implementation of the same functionality
- *Test artifacts*: reusing test plans, test cases and scenarios, and test data

On the other hand, in SOA, services are intended to be reusable building blocks and units of sharable software assets for different applications which implement different business processes. As a consequence, services can be orchestrated to construct composite services through business processes.

Core assets in SPL including a generic architecture and components develop applications whereas services are basic building blocks to support compositional software development in SOA in which business processes or application templates specify entire application through the definition of execution sequences of valid workflows. Services can be reusable artifacts which enable rapid SOA application development [54].

Assets and applications are generally tightly coupled in SPL, while services are loosely coupled which is one of the most pronounced properties of services in SOA research [46]. Services are highly independent of context and the state of other services.

Software components often operate within a context defined by a generic architecture for product family members in SPLs. SOA is grounded on the idea of open integration of business processes by means of shared services where services are described through standard-interface and are intended for reuse in different contexts. Nevertheless, services can also be developed and reused for internal processes within organizations. In essence, SOA basically envisages and focuses on large scale reuse [27] because SOA promotes services to be seamlessly consumed by diverse applications where they can be published, discovered and invoked through standardized specifications [6].

Unlike core assets, services can be reconfigured at runtime [9].

3.3 Architectural aspects of SPL versus SOA

Both SPL and SOA require defining the architectural context and composition rules with SPL architecture often centralized, static, and specialized into concrete products but SOA as decentralized. Composition rules are predefined in SPL, which describe common and variable behavioral characteristics of architecture, while in SOA composition or business rules are generally defined to govern the way in which a composition is constructed. SPL basically aims at providing a common architecture for reuse, whereas SOA lacks enough support for large grained software reuse at the architectural level. Gomma et al. [24, 51] discuss software architectural issues in SOA and describe various practices to develop reusable services in order to craft and compose systems from services efficiently. They draw attention that the architectural solution space offered by SOA promises to provide potentially significant benefits for reutilization. However, achieving SOA's benefits may not be guaranteed just by implementing based on the SOA solution. Accordingly, the important software architecture and reuse issues should be addressed prior to creating a SOA [51]. Tsai et al. propose a classification schema of architectures for SOA-based applications in order to evaluate variety of architectures [54]. The slackly coupling characteristic and platform-independent view inherited in SOA may address many architectural issues that are open-design and integration problems. Furthermore, architecture style offered by SOA is potential to maximize reuse beside interoperability and flexibility; however, SOA lacks support to manage variability at the architectural level [13, 35] whereas SPL enables managing variability to improve reutilization at such level.

3.4 Variability in SPL and SOA

The concept of *variability* refers to the ability of software systems or artifacts to be efficiently extended, modified, specialized, or configured (customized) for *(re)use* in the specific context for a particular application [56]. This characteristic enables for applying changes at different levels from software architecture to implementation. Two important concepts related to variability discussed in the literature are variation points and variants with the former being places in the design or implementation at which variants occur and with the latter being the alternatives to be selected at those variation points [47, 56].

Therefore, variability can specify a part of an architecture which remains variable, as variation points, or what is not completed at design time. Variability can be implemented at design time or run-time [52]. It is noteworthy that variability and flexibility are closely interconnected. Flexibility offers adaptation and changes of architecture, while variability deals with various version of architecture.

Variability in SPL encompasses all software artifacts from requirements to code [56, 12]. Therefore, there are numerous modeling methods proposed with the objective of modeling variability within software artifacts and at different levels of abstraction. Van Gur et al. discuss about the notion of variability in SPL [56], where

variability is exposed at different levels: platform technologies and user expectations, requirements specifications, designs, component source code, compiled code, linked code, and running code in the context of which variability refers to the ability to select among these artifacts at various stages.

Effective management of variability, which determines how flexibly new members of a given SPL can be obtained and defines SPL boundaries is essential for the success of SPLs [56]. The distinction between variability modeling and other techniques is based on the diversity between variability modeling and variability mechanism. Variability modeling techniques model the variability provided by the product line artifacts while variability mechanisms, several of which have been proposed in the literature such as conditional compilation, patterns, generative programming, macro programming, and aspect-oriented programming are commonly considered ways to introduce or implement variability in those artifacts.

Accordingly, variability in SPL is an essential concern in all phases of development life-cycle. Variability identification, modeling and management is rather a large field of research in SPL [11]. Most current works address identification and management of variability by modeling the concepts as *features* which considered as the first-class representation of variability and in terms of which the major advantages of discussing a software system is that the concept of feature bridges the gap between the requirements and technical design decisions in view of the fact that software components rarely address a single requirement but rather an entire set of essentials (details are given in Sec. 5). There are number of well-studied feature-oriented approaches for domain analysis and modeling common and variable requirements in SPLE such as FODA (Feature-Oriented Domain Analysis) [28] and its extension FORM (Feature-Oriented Reuse Method) [29], RSEB (Reuse-Driven Software Engineering Business) [25], GPM (Generative Programming Methods) [16] and PLUSS (Product Line Use case modeling for Systems and Software engineering) [19]. Every method generally shares feature as the common concept used in the analyses of commonality and variability. Some approaches are architecture-centric such as Hoek [55], Koalish [4], and Thiel [53]. Some of which are configuration-based, e.g., COVAMOF [48] and Koalish [4]. Some of which extend UML to model variability like VPM [58]. Some proposed approaches focus on separating variability representation from the representation of various SPL artifacts such as Bachmann [5].

The development of SOA-based applications is accomplished through different abstraction layers: business process or orchestration layer, service interface layer, and service implementation or component layer with the business process layer or orchestration consisting of composite services implementing coarse-grained business activities, or even an entire business process [45]. The service layer is composed of self-contained and business-aligned services which provide the implementation for fine-grained business activities. The service interface layer comprises the interface of services published by a service provider. Finally, the component layer (i.e., implementation layer) consists of a set of components that realize service interfaces and provide the implementation for services. Variability in SOA affects these different layers thoroughly. Chang et al. [9] discuss four types of variation points

which occur in a general four-layered SOA architecture: workflow variability, composition variability, interface variability, and business logic variability with workflow variability identified as variation of the control flow of a business process, i.e., tasks to be alternatively and optionally completed in a workflow depending on the individual service user, composition variability identified as variability when there is more than one possible service interfaces for activity construct in the business process which implement the service with either different logic or quality attributes and with Interface variability occurring when the candidate services interfaces are different. Finally, components which realize and implement service interfaces by different logic impose logic variability.

Granularity Levels: Granularity in SPL refers to the degree of detail and precision of variability as produced by design or implementation artifacts. SPL variability may exist at different levels of granularity ranging from entire components to single lines of code [30, 16]. SPLE takes a top-down approach and decomposes artifacts into fine grained artifacts whereas a bottom-up compositional approach is often adopted in SOA to combine artifacts into larger entities- inserting service into composite services (i.e., business processes) that finally form the application. Decomposition or top-down modeling means that an SPL architecture specifies the decomposition of a family into architectural components. However, there are also hybrid approaches, such as product populations modeled using Koala [57] where the mixture of bottom-up and top-down approaches are leveraged. In SOA, generally there is no particular architecture specifying the decomposition.

In SOA, granularity specifies the scope of variability in functionality exposed by a service. A component which provides an implementation for a service interface can be of various granularity levels that software developers can always encapsulate the entire functionality of a solution into a single service is possible due to the well-known 'fractal' nature of services, where a higher-level service can encapsulate lower-level services to any level of granularity [8]. However, a fine-grained service is more easily reused; in distinction, coarse-grained service is more difficult to be reused [51, 45]. Nevertheless, services with high-level interfaces increase the reusability because providing interfaces with a coarse-grained granularity masks specialized or implementation-specific methods, thereby, making a service adoptable and reusable by multiple applications. Moreover, creating and designing high-level, coarse grained interfaces that implement a complete business process is desirable from the perspective of service-oriented design and development [45, 20]. However, there is a trade-off between fine-grained and coarse-grained.

Services at different levels of granularity can be generally classified into different categories [33]: basic services, intermediary services, process-centric services and public enterprise services.

Basic services that represent the elements of a vertical domain are simple logic-centric or data-centric services with data-centric services handling persistent data and logic-centric services encapsulate algorithms for complex calculations or business rules and intermediary services designed to bridge a technical gasp in architecture. They provide service links with other services or application front-ends and services in gateways, adapters (mapping message formats to enable interop-

erability), facades (providing a different view on one or more services), and other functionality-adding services (extending functionality of existing services without altering them internally). In SOA, process-centric services control and maintain the state of the enterprises business processes which uses basic or intermediary services to perform task and deal with business data. These services separate process logic from representation layer and encapsulate the process complexity for a single point of administration. A common example is an online shopping process which includes filling the shopping cart, ordering products, and executing billing. Public enterprise services offered to partner companies as an in-house-system interface which, in turn, have the granularity of business documents and are coarse-grained integrate enterprises (B2B).

4 Running Example

To illustrate the concepts and the approach presented in the following sections, we select a part of case study of a family of online marketplace portals providing applications for online trading like eBay[1]. The portal, as an SOSPL, can be customized and deployed based on different business requirements of targeted stakeholders. Fig.1.(a) presents a service scenario of e-payment processes- part of a large product family that defines a common framework for online payment provided in online marketplace. For the simplicity, a high-level view of the payment process is represented, and the details are omitted.

Different methods of online payment can be considered for different instances of products from a family. Therefore, the number of possible payment method variations of a reference payment process, as a catalog and template, can be derived and customized according to the stakeholders' requirements and business objectives. Some services are indispensable and prerequisite of the payment process (e.g., `Credit Card` payment feature as the dominant online payment), which should be included for all the stakeholders' service product instance whereas some functional services (for example `Smart Card` `e-Check` and `Debit Card`) or extra-functional services (for instance `Logging` and `Monitoring`) can be determined as optional that can be included or excluded based on stakeholders' needs (See Fig.1.(b)). As a case in point, `Stakeholder A` may require additional features for having highly-secured payment transactions, incorporating a fraud protection service, even though this service is not required in the payment process of the final customized portal for `Stakeholder B`. In another scenario, `Stakeholder C` could ask the payment process to be supported by a `Mobile-based Notification` service in addition to the common payment notification services such as the `Mail-based Notification` service. Therefore, in the context of a product family, a business process should be imposed inevitably by variants (optional and mandatory services) which are required

[1] `http://www.ebayinc.com/`

to be managed, specialized and customized in order to meet different stakeholders' functional or quality requirements.

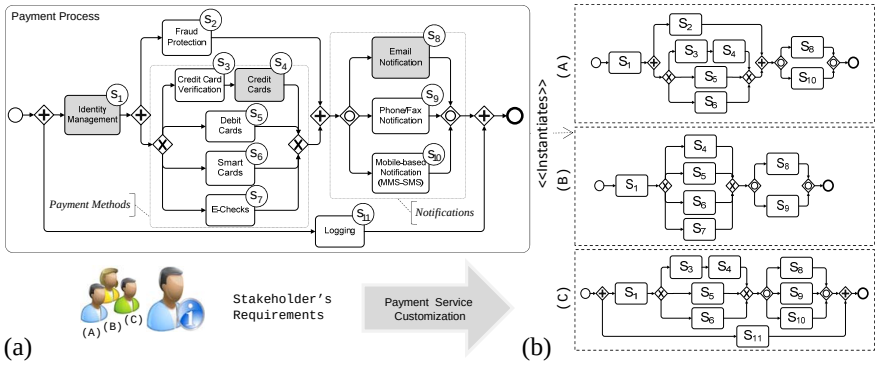

Fig. 1 a) A holistic view of e-Payment process family. b) e-Payment Process variants example

5 Applying SPLE for Development of Service-Oriented Software Product Lines

It is already mentioned that even though SOA has been widely adopted, there are still no systematic methods to support modeling and managing variability during the development of SOA-based applications and further service management. , which calls for a well-defined development process and understanding variability in functional and non-functional requirements in the course of development.

This section outlines the activities of a proactive methodology. The proposed method is an extension of a traditional software product-line life-cycle in order to support development and customization of a family of SOA-based applications. The proposed top-down method follows a two-life-cycle approach that separates two core activities related to *Service-Domain Engineering* and *Service-Application Engineering* (See Figs. 2 and 6). Service-domain engineering constructs and evolves the reuse infrastructure by analyzing the requirements and scoping the product line as a whole and producing any common, reusable business processes and services. On the other hand, service-application engineering derives individual services (i.e., customized services) from the reference architecture. Domain and application engineering life-cycles can rely on fundamentally different processes, namely, plan-driven and agile methods. In the following, we describe the major activities and their artifacts for three major development phases: (1) analysis (requirement engineering), (2) design, and (3) implementation and testing.

5.1 Service-Domain Engineering

The overall service-domain engineering processes of an SOSPL is depicted in Fig. 2. These activities (D1-D6 in Fig. 2) are performed iteratively. Domain analysis in SOSPL mainly encompasses product-line requirements engineering stage (D1) along with the analysis of variability by using feature modeling (D2). A feature model, as software artifact outcome of the feature modeling process, include the knowledge of variability associated with the functional and non-functional requirements and describes the permissible configuration space further guiding the customization process and determining how the reference business process model should be tailored according to the stakeholders' requirements in the application engineering life-cycle. During the domain design phase (D3), a reference business process model (also known as business process family) is designed and constructed for the product line architecture based upon the outcomes of the requirement engineering phase (D1). The model mapping (D4) establishes the mapping relationships between the features within the feature model and the corresponding activities specified within the reference business process model. The activities of the reference business process are delegated to the service(s) in SOSPLs. Inasmuch as non-functional (quality) requirements may also vary for different stakeholders, variability in the quality properties of services should also be captured and specified during the construction of an SOSPL (D5). To this end, features in the feature model are annotated by quality ranges which are supported by the entire product line architecture progressively helping service engineer and developers to evaluate the impact of variant features selected according to the quality characteristics that services provide [39]. In the final phase, the reference business process model is realized and implemented either by binding to the existing services or by developing new services. In the following, we detail these activities.

5.1.1 Product Line Requirements Analysis:

Similar to traditional requirements engineering, domain requirements engineering should at least include the following activities [50]: 1) *elicitation*, in which the product line business goals and stakeholders' requirements are discovered and scoped; 2) *specifications*, in which the requirements are analyzed in detail; 4) *validation*, in which the requirements are validated and consistency and completeness are checked, and 4) *management*, in which the requirements can be managed in terms of changes or refinements. In addition to these activities, domain requirements engineering captures commonality and variability between the requirements of several stakeholders. Moreover, an important activity of the requirements analysis of an SOSPL is to define the product line scope [13, 47, 43] and decide on the boundary of the product line.

A successful scoping which is determined by factors such as the knowledge of similar domain services and the future demands of stockholders is required to be performed carefully because a scope - either too large or too small - will impair the

capability of a SOSPL in achieving the goals of stockholders [12]. A goal-oriented domain analysis can be employed at the early stage of the requirement analysis in order to capture the product line goals for requirement elicitation and to further align the final service products with the business goals and intentions of both the stakeholders and service providers. This is accomplished at the different levels of abstraction by goal modeling about which interested readers can further read in [3]. The outcome of this phase is the requirements models which can be described by goal models, use-cases, documentations and details, which are used subsequently for the variability analysis of the product line under development.

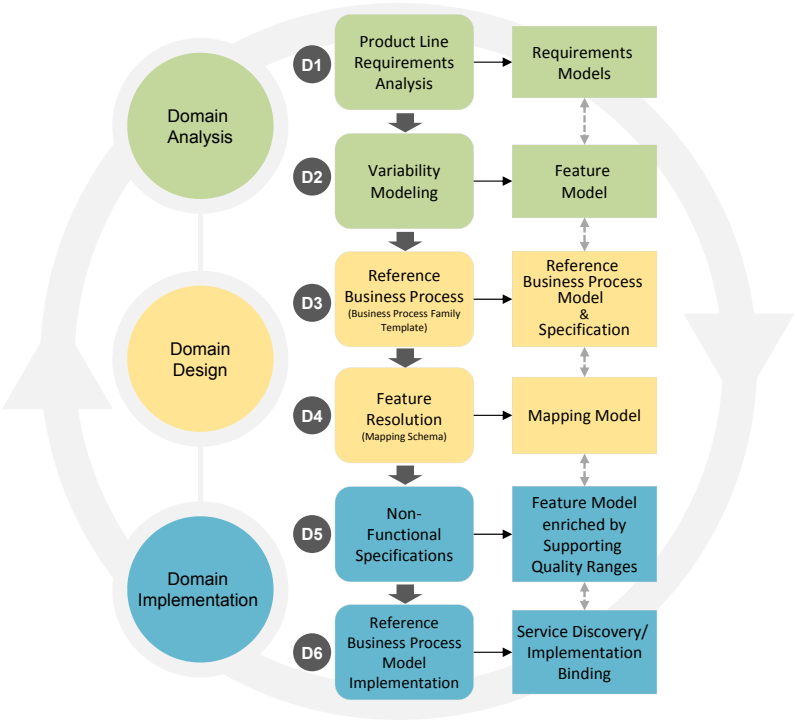

Fig. 2 Service-Domain Engineering of an Service-Oriented Software Product Line

5.1.2 Variability Analysis and Modeling:

The product line requirement engineering activity follows the variability analysis and modeling of the entire family in order to identify common and variable features. A *feature* is commonly defined as a visible incremental functionality and quality in software system(s) [28]. Nevertheless, depending on the stage of development

it may also refer to a requirement or a coarse-grained or fine-grained component in the system(s) which provide the required functionality from different technical views. The emphasis in the variability (i.e., feature) analysis is on optional features because optional features substantially differentiating one member of the family from the others.

In SOA, services constituting the orthogonal concept to the components notion, encapsulating functionality, and providing individual non-functional properties (i.e., QoS) through a well-described and published interfaces are characterized as building blocks of software that are loosely-coupled. From this view in the context of SOSPL, we define a feature as an increment in service functionality [1], which reflects stakeholders' both functional and non-functional requirements. Therefore, a feature, based on the granularity levels, can be realized and associated to a composite service at the high-level business processes, or associated and realized by an atomic service at the lower-level.

Feature-oriented development [28, 29, 16] is widely employed as a means for analysis, management, and visualization of commonality and variability in SPLE in terms of features at different abstraction levels. Feature modeling essentially organizes features of a software product family into a model called *feature model* residing between the requirement model and the design specification model (i.e., the reference business process model described in Sec. 5.1.3). Fig.3 shows a part of a feature model representing the variants (namely optional and alternative features) that characterize a requirements model. These features are selected to derive service products during the application engineering. Moreover, this model serves as a catalog of the variability space offered by a product family to accommodate the idiosyncrasies of the stakeholder enterprise or company.

Feature Model: A feature model consists of both formal semantics and graphical representation (e.g., feature diagram) and encompasses the knowledge of configuration for a product line. A feature model is a hierarchical decomposition of features in terms of parent-child relations on different levels of abstraction. Some of the features are not assumed to be present in every product during the application engineering, this differentiation is expressed by the indication of feature types and their relationships. Contrary to a mandatory feature always being selected if its parent is selected, an optional feature may or may not be selected. For instance, in Fig. 3, all the products should include the `Credit Card` feature as a mandatory feature. Other payment methods Fig. 3 are specified as optional features.

A feature cardinality and group cardinality can also be determined in cardinality-based feature modeling [17]. A cardinality defined as an interval, from zero to a given value and associated to a feature determines the lower and upper bound of the number of features required in any product in a product family. In the SOSPL context, this attribute specifies the number of service instances that should be linked to a given service at run time.

Or feature groups with defined cardinality indicate that at least k and at most k' features that can be included out of the n features ($k \leq k' \leq n$) in a group if the parent is selected. Moreover, *Alternative feature groups* with specified cardinality indicate that that only k out of n features in the group must be included if the parent

is selected. Back to the simplified feature model example from Fig. 3, all the products should include the `Payment Notification` features. Also, all the final derived service products should include at least two methods of notification according to the feature model.

Furthermore, because features are not always independent, *integrity constraints* (i.e., the *includes and excludes*) can be defined over features of a feature model to model dependencies and relations which can exist among them. They are the means to describe that the presence of a certain feature in the product imposes the presence or exclusion of another feature (See Fig.3).

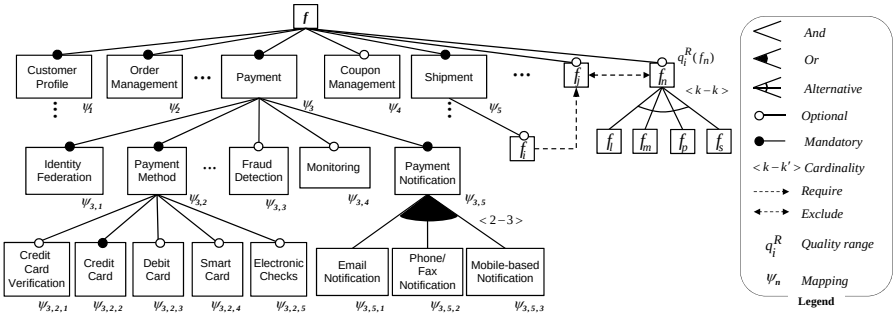

Fig. 3 Feature model of e-Payment (conforming stakeholders' requirements model)

Feature models are an efficient abstraction of variability derived from the domain and stakeholders' requirements. They also help to derive the design and the development of variability through all the stages of the development including service identification and design, and further customization [13, 40].

In the feature-oriented analysis phase, which subsequently guides the identification of candidate services with right granularity, we organize feature based on the following criteria:

- Features supporting a particular business process can be grouped and abstracted as a higher-level feature on a coarse-grained level (e.g., `Payment`)
- Features supporting specific functional or non-functional services can be grouped and abstracted as a higher-level feature (e.g., `Payment Notification and Logging` services)
- A feature which incrementally realizes a feature at the upper-level, subsequently becomes as a sub-feature at the lower-level
- Features at the leaf-level are realized by fine-grained services

5.1.3 Reference Business Process Model:

The previous activities, domain decomposition, and top-down variability analysis and modeling provide an insight into product features of a target domain. A feature model is generated as an output of the domain analysis. This model, which is generated as an output of the domain analysis, is then used to derive reference architecture and develop reusable components (business processes and services) in the course of the domain design. The design phase produces an architecture-interdependent model defining reference architecture as the behavioral model of features for the entire family and specifies how features are composed at run-time.

A template-based approach has been widely adopted in SPLE to create the reference model. In the case of SOSPLs, such a reference model is designed as a template for the entire service products family in a superimposed way [15]. A reference business process model, as a model template, describes and specifies the execution sequence of services for all instances of the product line. That is, a reference business process model is a union of all the business processes of the product line and 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 process-oriented modeling languages (e.g., BPMN, EPC, and/or YAWL), and incrementally refined and optimized. For example, Fig.4 illustrates a part of reference business process model, where variability and configuration knowledge have been modeled and encapsulated in given feature model in Fig. 3.

The reference business process model configured through the selection/elimination of features from the feature model during the application engineering and executive instances are derived (See Fig. 1.(b)). In other words, according to the fact that architectural variations in the reference model are encoded as features, various parts of the reference business process model are organized in variation points. These variation points are managed and configured by means of feature models. To point out the differences between design and runtime variability, it should be mentioned that feature models capture and encapsulate only architectural variability at design time. In contrast, business process models describe behavioral variability, i.e., how features are composed, which drives runtime variability through composition patterns (discussed in the next section).

Furthermore, feature model configuration (i.e., specialization and customization) is performed during the build time. The configuration can be done through the process of staged-configuration [18] where features further are prioritized and selected according to the (non-)functional requirements of the stakeholders [40]. All configured service products, which are instances of the family, have to conform to the reference architecture.

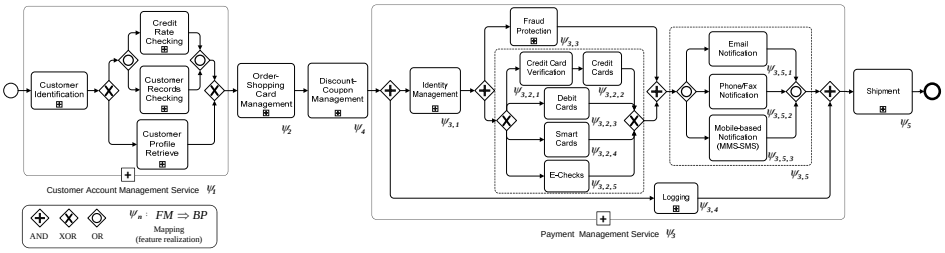

Fig. 4 A part of reference business process model

5.1.4 Feature Resolution and Mapping Model:

During the design phase, feature resolution is the analysis and connection of the feature model and the reference business process model in order to specify explicit mapping links between the two models: feature and reference business process model, the outcome of which constitutes a mapping model including links between features in the feature model activities in the reference business model. This mapping model enables configuration of the reference business process model through feature section during application engineering. From a certain point of view, this mapping model also provides the traceability links between the requirements and implementation [15, 49].

A mapping model can simply consist of boolean expressions specifying presence or removal of a modeling element (e.g., activity (abstract service)) in a model template (i.e., a reference business process model in our case) based on the selection of features in the feature model [15]. In our approach, we consider a boolean variable ψ_i corresponding to each feature f_i. This approach uses *presence conditions* (PC) as annotation properties for each activity within the reference business process model. The PC of an activity is formulated as a boolean expression of ψ_i variables corresponding to the features mapped to the activity (See Fig. 3 and 4). Both the feature and activity constructs refer to model elements of feature models and reference business process models. Thereby, when domain engineers map features to activities, the PCs of activities are defined. In application engineering, when a feature f_i are removed from the configuration, its corresponding ψ variables are set to *false*.

Feature resolution also helps to identify cross-cutting concerns related to general non-functional requirements. For example, feature `Monitoring` with given mapping annotation ψ_i in Fig. 3 is mapped to activity `Monitoring` as an extra-functional abstract service in the reference business process model (see Fig. 4). Based on the selection of features from the feature model in application engineering, the reference business process model is configured (Fig. 1.(b)).

5.1.5 Non-functional Specifications:

The domain design phase is also followed by the specification of non-functional properties based on the non-functional requirements (NFRs) analysis because NFRs are interlaced and related to functional requirements. Variability in NFRs influences the SOSPL design and implementation. Non-functional variations often exhibit different types and levels of quality properties (e.g., normal and strong authentication or security). For instance, NFRs for feature `Credit Card` can include *cost, security, availability* and *reliability* or they can also entail defined domain-specific non-functional aspects such as *usability* and *convenience of use*. Furthermore, in application engineering, non-functional variations directly impact the selection of appropriate services from candidate services, all of which provide equivalent functionalities even though with different degrees of non-functional properties related to the service quality specification. To this end, there are a number of proposals [7], in which feature models are extended to support feature attributes to comprise non-functional properties which can be measured (or to be measured) such as cost, availability, latency, bandwidth, etc).

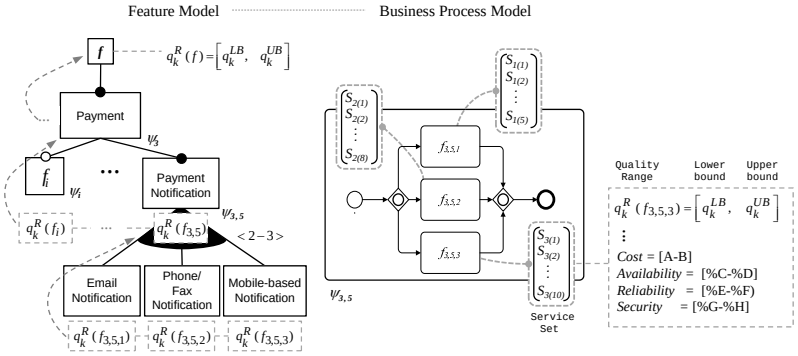

Fig. 5 Non-functional specification and aggregation for evaluating quality range supports by product line architecture

In the context of SOSPL, these non-functional properties can be viewed as QoS properties, which are associated with each feature. Mapping models interconnecting feature and business process models enable propagation of quality property values of concrete service sets, bounded to activities (abstract services) within in the process model. Based on the underlying implementation of a set of functionally equivalent services, which may be available for each feature, ranges of values of quality properties can be further specified and aggregated for each feature. Particularly during the domain engineering life-cycle, determining the implied QoS ranges for individual feature helps domain engineers ensure that the product line architecture will fulfill and deliver the upper and lower bounds of the values of the quality requirements requested by the stakeholders. Moreover, quality range computation enables for keeping track of the product line quality ranges even after the specification of the

service quality has changed. For example, in Fig. 5, sets of candidate services provide different range of quality, denoted as q^R, for each features. The range of the k^{th} quality property for feature f_i can be hierarchically computed. In [39], we introduce a generic evaluation model and method for aggregation and computation of ranges of quantified values of quality properties defined for product line architectures.

5.1.6 Reference Business Process Model Implementation:

The domain design phase produces a reference business process model and architecture for a family of service products (i.e, SOSPL). In the domain implementation and realization, the reference business process model is realized and implemented, which involves implementing and testing the detailed architecture of the family modeled by reference model. Abstract services specified by the reference business process model are implemented using component models such as Java class, Enterprise Java Beans, or .Net components. However, some of the services needed for the implementation might already be available; for instance, it can either be found in service catalog or retrieved through a service discovery process, and some of the services could potentially be built by partly reusing or modifying existing solutions.

5.2 Service-Application Engineering

This section describes a holistic view to the application engineering life-cycle which includes the major phases of service customization and derivation from the business process family. Regardless of the chosen variability modeling approach, the ultimate of in-service-application engineering is to employ the variability defined in domain engineering by selecting shared assets similarly developed in domain engineering. Fig. 6 depicts a high-level application engineering process which starts with the elicitation and capturing of both the functional and non-functional requirements of an individual stakeholder through the phase of application-requirements analysis (A1). In the application design phase (A2), features are prioritized based on the stakeholder's captured preferences and business objectives concerning the optional features and quality needs. Thereafter, the feature model is specialized through the decision-making process of selecting optional features. Subsequently, the reference business process model is configured and corresponding services are selected and bound in the deployment and integration phase (A3). The details of these application engineering phases can be found in [40].

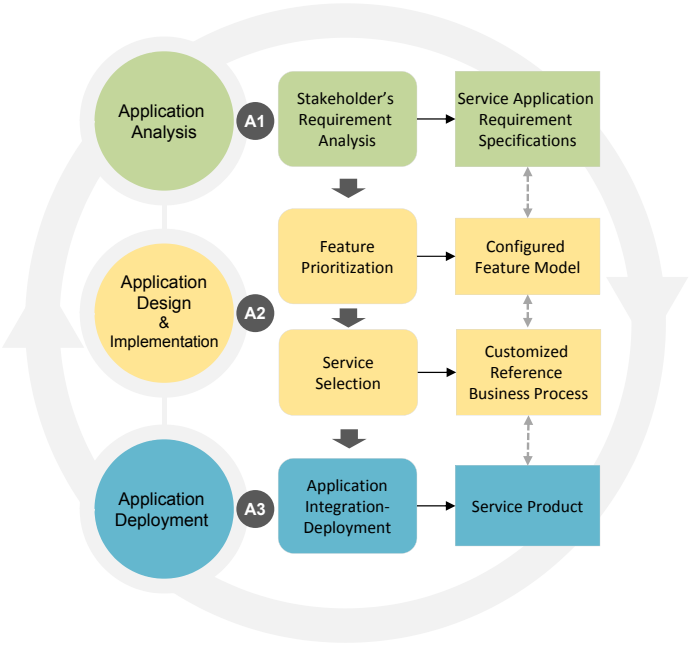

Fig. 6 Application Engineering of an Service-Oriented Software Product Line

5.2.1 Application Analysis:

This phase focuses on the elicitation of requirements of a particular stakeholder to derive customized process variants, which can be deployed as the final product. The preferences of the stakeholder are captured and later utilized for feature prioritization and selection. Similar to the requirements engineering phase in service engineering methodologies like SOMA [2], activities in the application analysis phase capture requirements for a single service (application); however, the application analysis activities reuse the family requirements models to develop requirements models of a target service. For example, assuming a family requirement model is represented in a goal model, the service goal model is developed through reasoning on the family goal model based on the inputs of current stakeholders [3, 60]. Validation and verification of the application requirements model with respect to the stakeholder's needs and product line constraints are performed. In the context of marketplace portals, stakeholders of a target service application may request payment, shipment, order management, and manage customer functionalities as well as high security and low cost. The detail of stakeholders requirements can be achieved by applying label propagation algorithms on the marketplace family goal model.

5.2.2 Application Design and Implementation:

During this phase, the feature model is utilized to manage and select variants constituting service product instances. This is accomplished through the feature prioritization and selection of sub-processes. This activity includes the selection of the best and permissible combination of optional features along with the selection of the corresponding services that would optimally satisfy the stakeholder's functional and non-functional requirements. Several automatic or semi-automatic and manual feature model configuration techniques have been proposed to guide the final product configuration (i.e., customization) according to the requirements and preferences of stakeholders. Automatic configuration approaches employ AI optimization techniques such as Genetic Algorithms (GAs), Bayesian Belief Networks, and Constraint Satisfaction Optimization Problem (CSOP) to create the final customized product [7]. On the other hand, manual configuration techniques through staged-configuration [18] provide specialization steps for service engineers and help them resolve variability in the process of family customization (see Sec. 6). After configuring the feature model, due to the established mappings between the feature model and the reference business process model, a concrete business process for a target service-oriented application and its realizations are derived from the family design and implementation models. However, since there may be some requirements which could not be satisfied by existing assets (i.e., services) contained in the developed SOSPL architecture, further refinement of instantiated service products from the reference model can be performed, and new required services can be implemented. In our running example, according to the requirements of the current application derived in the previous stage, application engineers can configure marketplace feature model and derive a business process model for the service-oriented application under development. Also, proper services based on the requested quality of services (e.g., high security and low cost) are selected.

5.2.3 Application Deployment:

This phase focuses on creating an executable business process and deployment of the customized services in the production environment after validating the customized services against the application requirement specification. After the deployment of the final service product on to the stakeholders' environment, the execution of the customized services is monitored to ensure the compliance of the service execution to stakeholders' requirements and any service level agreements.

6 Discussion

The development, management and evolution of many modern software systems rely on the notion of variability and suitable design techniques. SPLE research

has devoted a considerable amount of resources to the development of various approaches to deal with variability analysis, modeling, management, customization and related challenges over the last decade. These approaches can be employed in the design and development of variant-rich service-oriented applications (referred to as SOSPLs in this chapter).

Feature-oriented analysis is a means to create variability in services at different levels of abstraction and to subsequently make the managing and customizing it possible. Variability can be considered in terms of four different general levels of abstractions in service-oriented development [45]: requirements, business process models, service interface model and service component. In that sense, variability at a lower-level of abstraction realizes variability at a higher level. As described earlier in the chapter, we leveraged feature modeling for managing variability by focusing on the requirements and business processes at the higher levels of abstraction. However, feature models can also be employed to model, represent and manage variability at the levels of service components and interface to support efficient service management. For instance, in [21], Fantinato et al. employed feature modeling to manage and enable customization in service contracts.

Nguyen et al. [41] propose a feature-based service customization framework to model and manage variability of complex Web service specifications. The proposed approach employed feature models as an extension to service description artifacts in order to facilitate the customization of service interfaces. In [42], the same authors adopt a feature-oriented approach to model variability in process-based service compositions and to enable process customization. The approach extends the BPMN 2.0 metamodel to allow for defining variation points and variants within business process models. The extension is focused on modeling variability of three aspect of business process: control flow, data flow, and message flow. A variation point in a control flow is interpreted as any location in a process model at which different execution paths can take place, and variants can be arbitrary process fragments. Variability in data flow is considered as a different way for storing data objects. Variability in message flow is identified as alternative conversations and interactions between two parties, i.e., the process and a partner service (or a consumer).

Koning et al. [31] investigate how variability can be incorporated into service-based systems in order to enable variability modeling and management. They describe how variability management helps to support run-time reconfiguration of systems by service replacement corresponding to the non-functional requirements of stakeholders. VxBPEL is proposed as an extension of Business Process Execution Language (BPEL) to the process description and definition. VxBPEL allows for run-time variability and variability management in Web service-based systems. Variability information is defined in-line with the process definition. VxBPEL builds upon COVAMOF [48], a framework to model variability.Koning et al. note that the architectural modeling and management of variability in Web service-based systems provides the following advantages: enhancing the extensibility of systems through service replacement, improving run-time flexibility to configure and rebind services (e.g., being able to optimize quality through reconfiguration).

As already mentioned, improving reusability in service-oriented development is an often-stated goal in the literature. There are a number of important concerns that can influence highly-important analysis and design decision for the quality of service design. The major concerns include analysis and design for service reuse, service granularity management, and design of composable service [45]. Hence, it is a demanding feat to develop service-oriented systems such as how to identify reusable services at the right level of granularity to facilitate service composition and in turn, testifying that the identification of service candidate is a challenging task in service engineering [2, 45]. SPL approaches can be adopted to consolidate design principles and service identification during the course of service engineering.

Lee et al. [37] present a feature-oriented approach to the analysis, identification and development in order to improve reusability of service-based systems. The proposed approach provides guidelines about how to address the key issue of granularity and orchestration of services by using feature models. They show how reusable service can be identified and specified based on software features. The proposed method is based on analysis of features that may vary from a user's point of view and will be subject to reconfigurations at runtime. Another approach to using feature-oriented analysis for service identification during the analysis and design phases is proposed by Chen et al. [10]. They focus on re-engineering towards service-oriented systems and the remark of whom that feature-oriented analysis bridges the gap between the abstract architectural and source code level, whereas business processes are excluded.

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. The authors of SOMA remarks that variability analysis in the practical SOA solution design is crucial for the initial finding-binding relationships between a service consumer (i.e., stakeholder) and a service provider. Moreover, it was noted that the publishing and discovery of relationships are often affected by variations, which are identified later in the design process. Hence, such variations may cause expensive fundamental re-design of SOA-based solutions [2]. To address this problem, the authors remark that a development life-cycle for SOA-based solutions should be extended by a variation-oriented analysis as an extra dimension that should be performed.

7 Conclusion

We can observe that the convergence of service-oriented and software product line engineering is gaining a considerable amount of attention and rapidly emerging as a viable and important software development paradigm. As we have discussed in this chapter, they both share common goals and promises to collaborate in the development of flexible, cost-effective software systems and to support a high level of reuse. Yet, their main goals are somewhat different. In this chapter, we discussed

how service-oriented development can benefit from SPLE approaches for variability modeling and management in the process of identification and design of variant-rich service-oriented applications.

By combining ideas of service-oriented development and SPLE, we expect to derive new software engineering approaches to make use of the best from both paradigms: (a) development of generic software architectures for highly adaptive Web services that can respond effectively to fluctuations in stakeholders' (non-) functional requirements, and (b) development of shared architectures that could be reused in different instances (benefits from the SPLE principles).

References

1. S. Apel, C. Kaestner, and C. Lengauer. Research challenges in the tension between features and services. In *Proc. of the 2nd Int. workshop on Systems development in SOA environments*, pages 53–58. ACM, 2008.
2. A. Arsanjani, S. Ghosh, A. Allam, T. Abdollah, S. Ganapathy, and K. Holley. Soma: A method for developing service-oriented solutions. *IBM Systems Journal*, 47(3):377–396, 2008.
3. M. Asadi, E. Bagheri, D. Gašević, M. Hatala, and B. Mohabbati. Goal-driven software product line engineering. SAC '11, pages 691–698, NY, USA, 2011. ACM.
4. T. Asikainen, T. Soininen, and T. Männistö. A Koala-based approach for modelling and deploying configurable software product families. *Software Product-Family Engineering*, pages 225–249, 2004.
5. F. Bachmann, M. Goedicke, J. Leite, R. Nord, K. Pohl, B. Ramesh, and A. Vilbig. A meta-model for representing variability in product family development. *Software Product-Family Engineering*, pages 66–80, 2004.
6. B. Benatallah and H. M. Nezhad. Service oriented architecture: Overview and directions. In *Advances in Software Engineering*, volume 5316 of *Lecture Notes in Computer Science*, pages 116–130. Springer Berlin / Heidelberg, 2008.
7. D. Benavides, S. Segura, and A. R. Cortés. Automated analysis of feature models 20 years later: A literature review. *Inf. Syst.*, 35(6):615–636, 2010.
8. C. Bussler. The fractal nature of web services. *Computer*, 40:93–95, March 2007.
9. S. H. Chang and S. D. Kim. A variability modeling method for adaptable services in service-oriented computing. In *SPLC '07: Proc. of the 11th Int. Software Product Line Conference*, pages 261–268, DC, USA, 2007. IEEE Computer Society.
10. F. Chen, S. Li, and W. C.-C. Chu. Feature analysis for service-oriented reengineering. In *APSEC '05: Proc. of the 12th Asia-Pacific Software Eng. Conference*, pages 201–208. IEEE Computer Society, 2005.
11. L. Chen, M. Ali Babar, and N. Ali. Variability management in software product lines: a systematic review. In *Proc. of the 13th Int. Software Product Line Conference*, pages 81–90. CMU, 2009.
12. P. Clements and L. Northrop. Software product lines: Practices and patterns. 2001.
13. S. G. Cohen and R. Krut. Managing variation in services in a software product line context. Technical Report SEI-2010-TN-007, CMU, May 2010.
14. S. G. S. Cohen and R. W. Krut. Proc. of the 1st workshop on Service-Oriented architectures and product lines: What is the connection? Technical Report CMU/SEI-2008-SR- 006, 2008.
15. K. Czarnecki. Mapping features to models: A template approach based on superimposed variants. In *GPCE 2005*, pages 422–437. Springer, 2005.
16. K. Czarnecki and U. Eisenecker. *Generative Programming: Methods, Tools, and Applications*. Addison-Wesley Professional, June 2000.

17. K. Czarnecki, S. Helsen, and U. Eisenecker. Formalizing cardinality-based feature models and their specialization. *Software Process: Improvement and Practice*, 10(1):7–29, 2005.
18. K. Czarnecki, S. Helsen, and U. W. Eisenecker. Staged configuration through specialization and multilevel configuration of feature models. *Software Process: Improvement and Practice*, 10(2):143–169, 2005.
19. M. Eriksson, J. Börstler, and K. Borg. The pluss approach–domain modeling with features, use cases and use case realizations. *SPLC*, pages 33–44, 2005.
20. T. Erl. *Service-oriented architecture: concepts, technology, and design*. Prentice Hall PTR Upper Saddle River, NJ, USA, 2005.
21. M. Fantinato, M. B. F. de Toledo, and I. M. de Souza Gimenes. Ws-contract establishment with qos: an approach based on feature modeling. *Int. J. Cooperative Inf. Syst.*, 17(3):373–407, 2008.
22. W. B. Frakes and K. Kang. Software reuse research: Status and future. *IEEE Trans. Softw. Eng.*, 31:529–536, July 2005.
23. M. Galster. Describing variability in service-oriented software product lines. In *Proc. of the Fourth European Conf. on Software Architecture: Companion Volume*, pages 344–350, 2010.
24. H. Gomaa. Advances in Software Design Methods for Concurrent, Real-Time and Distributed Applications. In *Software Engineering Advances, 2008. ICSEA'08. The 3rd Int. Conf. on*, pages 451–456. IEEE, 2008.
25. M. L. Griss, J. Favaro, and M. d. Alessandro. Integrating feature modeling with the rseb. In *Proc. of the 5th Int. Conference on Software Reuse*, ICSR '98, pages 76–, Washington, DC, USA, 1998. IEEE Computer Society.
26. A. Helferich, G. Herzwurm, S. Jesse, and M. Mikusz. Software product lines, Service-Oriented architecture and frameworks: Worlds apart or ideal partners? In *Trends in Enterprise Application Architecture*, pages 187–201. IEEE Computer Society, 2007.
27. M. Huhns and M. Singh. Service-oriented computing: Key concepts and principles. *IEEE Internet Computing*, 9(1):75–81, 2005.
28. K. Kang, S. Cohen, J. Hess, W. Novak, and A. Peterson. Feature-oriented domain analysis (FODA) feasibility study, 1990.
29. K. Kang, S. Kim, J. Lee, K. Kim, E. Shin, and M. Huh. FORM: a feature-;oriented reuse method with domain-;specific reference architectures. *Annals of Software Engineering*, 5:143–168, 1998.
30. C. Kästner, S. Apel, and M. Kuhlemann. Granularity in software product lines. In *Proc. of the 30th Int. Conf. on Software engineering*, pages 311–320. ACM, 2008.
31. M. Koning, C. Sun, M. Sinnema, and P. Avgeriou. VxBPEL: Supporting variability for Web services in BPEL. *Inf. and Software Technology*, 51(2):258–269, 2009.
32. W. Kozaczynski and G. Booch. Component-based software engineering. *IEEE software*, 15(5):34–36, 1998.
33. D. Krafzig, K. Banke, and D. Slama. *Enterprise SOA: service-oriented architecture best practices*. Prentice Hall PTR, 2005.
34. R. W. Krut and S. G. Cohen. 2rd workshop on service-oriented architectures and software product lines: Putting both together. pages 115–147. CMU, 2009.
35. R. W. Krut and S. G. Cohen. 3rd workshop on service-oriented architectures and software product lines: Enhancing variation. pages 301–302. CMU, 2009.
36. J. Lee and G. Kotonya. Combining service-orientation with product line engineering. *IEEE Software*, 27:35–41, 2010.
37. J. Lee, D. Muthig, and M. Naab. A feature-oriented approach for developing reusable product line assets of service-based systems. *Journal of Systems and Software*, 83(7):1123–1136, July 2010.
38. J. McGregor, D. Muthig, K. Yoshimura, and P. Jensen. Guest editors' introduction: Successful software product line practices. *Software, IEEE*, 27(3):16 –21, 2010.
39. B. Mohabbati, D. Gašević, M. Hatala, M. Asadi, E. Bagheri, and M. Bošković. A quality aggregation model for service-oriented software product lines based on variability and composition patterns. In *ICSOC*, pages 436–451, 2011.

40. B. Mohabbati, M. Hatala, D. Gašević, M. Asadi, and M. Bošković. Development and configuration of service-oriented systems families. SAC '11, pages 1606–1613, NY, USA, 2011.
41. T. Nguyen and A. Colman. A Feature-Oriented Approach for Web Service Customization. In *2010 IEEE Int. Conf. on Web Services*, pages 393–400. IEEE, 2010.
42. T. Nguyen, A. Colman, and J. Han. Modeling and managing variability in process-based service compositions. In *Proc. of the 9th Int. Conf. on Service-Oriented Computing*, ICSOC'11, pages 404–420, Berlin, Heidelberg, 2011. Springer-Verlag.
43. L. Northrop. Sei's software product line tenets. *Software, IEEE*, 19(4):32–40, 2002.
44. M. Papazoglou, P. Traverso, S. Dustdar, and F. Leymann. Service-oriented computing: A research roadmap. *Int. J. of Cooperative Inf. Sys.*, 17(2):223–255, 2008.
45. M. Papazoglou and W. Van Den Heuvel. Service-oriented design and development methodology. *Int. J. of Web Engineering and Technology*, 2(4):412–442, 2006.
46. M. P. Papazoglou. Service-oriented computing: Concepts, characteristics and directions. *Web Information Systems Engineering, Int. Conference on*, 0:3, 2003.
47. K. Pohl, G. Böckle, and F. J. van der Linden. *Software Product Line Engineering: Foundations, Principles and Techniques*. Springer-Verlag New York, Inc., 2005.
48. M. Sinnema, S. Deelstra, J. Nijhuis, and J. Bosch. Covamof: A framework for modeling variability in software product families. *SPLC'04*, pages 25–27, 2004.
49. P. Sochos and M. Riebisch. Feature-oriented development of software product lines: Mapping feature models to the architecture. In *Object-Oriented and Internet-Based Technologies*, pages 138–152. Springer, 2004.
50. I. Sommerville and P. Sawyer. *Requirements engineering*. Wiley London, 1997.
51. J. Street and H. Gomaa. Software Architectural Reuse Issues in Service-Oriented Architectures. In *hicss*, page 316. IEEE Computer Society, 2008.
52. M. Svahnberg, J. Van Gurp, and J. Bosch. A taxonomy of variability realization techniques. *Software: Practice and Experience*, 35(8):705–754, 2005.
53. S. Thiel and A. Hein. Systematic integration of variability into product line architecture design. *Software Product Lines*, pages 67–102, 2002.
54. W. Tsai, Z. Jin, P. Wang, and B. Wu. Requirement engineering in service-oriented system engineering. In *e-Business Engineering, 2007. ICEBE 2007. IEEE Int. Conference on*, pages 661–668. IEEE, 2007.
55. A. van der Hoek. Design-time product line architectures for any-time variability. *Sci. Comput. Program.*, 53(3):285–304, 2004.
56. J. Van Gurp, J. Bosch, and M. Svahnberg. On the notion of variability in software product lines. In *Proc. of the Working IEEE/IFIP Conference on Software Architecture*, page 45, 2001.
57. R. van Ommering. Building product populations with software components. In *ICSE '02*, pages 255–265. ACM, 2002.
58. D. L. Webber and H. Gomaa. Modeling variability in software product lines with the variation point model. *Sci. Comput. Program.*, 53(3):305–331, 2004.
59. E. Ye, M. Moon, Y. Kim, and K. Yeom. An approach to designing Service-Oriented Product-Line architecture for business process families. In *Advanced Communication Technology, The 9th Int. Conf. on*, volume 2, pages 1002, 999, 2007.
60. O. Zimmermann, P. Krogdahl, and C. Gee. Elements of Service-Oriented Analysis and Design, June 2004.