Towards an Approach for Service-Oriented Product Line Architectures

Flávio Mota Medeiros[1,2] Eduardo Santana de Almeida[2,3]
Silvio Romero de Lemos Meira[1,2,3]
Federal University of Pernambuco (UFPE)[1]
Reuse in Software Engineering (RiSE)[2]
Recife Center for Advanced Studies and Systems (C.E.S.A.R.)[3]
{fmm2,srlm}@cin.ufpe.br esa@rise.com.br

Abstract

Service-Oriented Architecture (SOA) has appeared as an emergent approach for developing distributed applications as a set of self-contained and business-aligned services. SOA aids solving integration and interoperability problems and provides a better Information Technology (IT) and business alignment, giving more flexibility for the enterprises. However, SOA does not provide support for high customization and systematic planned reuse to develop applications that fit customer individual needs. In this paper, we propose an approach in which SOA applications are developed as Software Product Lines (SPLs). Thus, the term Service-Oriented Product Line is used for service-oriented applications that share common parts and vary in a regular and identifiable manner. In this context, high customization and systematic planned reuse are achieved through managed variability and the use of a two life-cycle model as in SPL engineering: core assets and product development. We conclude the paper with an initial case study in the conference management domain explaining the steps of our approach.

1. Introduction

In software development, there is an essential need to reduce costs, effort, and time to market of software products [1]. It is crucial to develop flexible systems able to adapt to market changes quickly [2]. In addition, there are lots of different technologies appearing, and enterprises need to integrate their software investments (legacy systems) with these new technologies [3]. However, the complexity and size of systems are increasing, and products must fit customer or market segment needs [4].

In this context, SOA is an emergent approach to solve integration and interoperability problems [5, 6], align IT and business goals, and increase business flexibility [2]. However, SOA lacks on support for high customization and systematic planned reuse. In other words, despite of the natural way of achieving customization in service-oriented applications, changing service order or even the participants of service compositions, services are not designed with variability to be highly customizable and reusable in specific contexts. In addition, service artifacts, e.g., specifications and models, are not designed with variability as well. Hence, these artifacts cannot be easily reused by a family of service-oriented applications [7].

Thus, SPL engineering, which has the principles of variability, customization and systematic planned reuse in its heart, can be used to aid SOA to achieve these benefits. In this path, service-oriented applications that support a particular set of business processes can be developed as SPLs [8, 9]. The motivation for it is to achieve desired benefits such as productivity gains, decreased development costs and effort, improved time to market, applications customized to specific customers or market segment needs, and competitive advantage [4, 10].

In this paper, we propose an approach for service-oriented product line architectures that combines SPL and SOA concepts and techniques to achieve high customization, systematic planned reuse and the desired benefits mentioned before.

Hence, the concept of managed variability and systematic planned reuse were introduced into service-oriented development activities. In order to deal with these concepts, the development process was divided in two life cycles as in SPL engineering [4, 11]. The first, *core assets development*, produces generic artifacts with variability to establish a production capability for applications. The second, *product development*, resolves the variation points of the generic artifacts produced in core asset development and creates applications customized to specific customers. Management at the technical and organizational levels during core assets and product development must be strongly committed to the success of the product line [12].

The reminder of this paper is organized as follows. Section 2 presents an overview of the approach for service-oriented product line architectures, and Section 3 describes its inputs, outputs and activities in details. A case study on the conference management domain is presented in Section 4. Related work is discussed in Section 5, and, Section 6 presents some concluding remarks and directions for future work.

2. Approach Overview

In this section, an overview of the approach for service-oriented product line architectures is presented. It is a top-down approach for the systematic identification, and documentation of service-oriented core assets supporting the non-opportunistic reuse of SOA.

The approach is based on the architectural style shown in Figure 1. This architectural style was adapted from [13, 14], which present a complete list of layers commonly used in SOA development. As mentioned, the architectural style is divided into layers, each of them with specific purposes as described next.

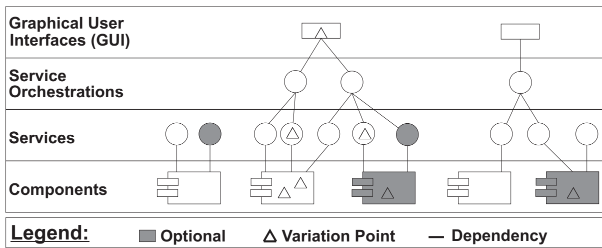

Figure 1. Architectural Style.

The *interface layer* is composed of Graphical User Interfaces (GUI) components. This layer may be used only by service-oriented product lines that require visual interfaces to interact with services and service orchestrations. The *orchestration service layer* consists of composite services, which implement coarse-grained business activities, or even an entire business process, that need the participation and interaction of several fine-grained services. The *service layer* is composed of self-contained and business-aligned services, which implement fine-grained business activities. Finally, the *component layer*, which consists of a set of components that provide functionality for the services exposed in the service layer and maintain their Quality of Service (QoS).

Note that the architectural elements (components, services, service orchestrations and user interface components) of these layers are developed with variability, and they can be mandatory, optional or alternative.

As mentioned previously, the approach is divided in two life cycles as in software product line engineering [4, 11]: *core assets* and *product development*. The *core assets development* aims to provide guidelines and steps to identify, document and implement generic architectural elements with variability. During *product development*, these architectural elements are specialized to a particular context according to specific customer requirements or market segments needs.

In this paper, we focus on the core assets development. In particular, on the design of domain specific architectures for service-oriented product lines. Thus, we provide guidelines and steps for the identification and documentation of components, services, service orchestrations and their flows using a top-down approach. In other words, the identification of architectural elements from existing legacy systems, the bottom-up approach, is not considered in this work. The following section presents the inputs, outputs and activities of the approach for service-oriented product line architectures in more details.

3. The Approach

The approach for service-oriented product line architectures starts with an identification phase. It receives the *feature model* and the *business process models* as mandatory inputs, and produces a list of possible components, service candidates and service orchestration candidates for the product line architecture. Thus, these architectural elements can be reused in all products of the line. This phase is separated in *component identification* and *service identification* activities.

Subsequently, there is a *variability analysis* activity. It receives the list of components and services identified previously, and defines and documents key architectural decisions regarding variability. In this activity, it is defined how the variability will be implemented within the services and components.

Architecture specification activity concludes the approach. In this activity, the architecture is documented using different views in order to represent the concerns of the different stakeholders involved in the project [15].

Figure 2 shows the inputs, outputs and activities of the approach for service-oriented product line architectures.

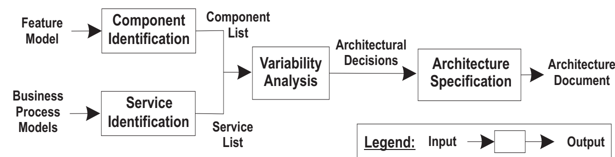

Figure 2. Activities of the Approach.

The next sections present the activities of the approach in more details. An initial case study clarifying and explaining these activities with examples is presented in Section 4.

3.1 Component Identification

In this activity, the components of the service-oriented product line will be identified. We consider a software component as a self-contained artifact with well-defined interfaces and subject to third-party compositions [16].

This activity starts with an analysis of the feature model to identify architectural component candidates. The purpose of this activity is to put features into modules (components) in order to design an architecture where components can be added or removed to generate customized products. Each of the modules identified in this activity will be an architectural component candidate for the service-oriented product line architecture.

In order to clarify this activity, we will use an alternative feature with two variants as example. In this case, each variant will be placed in a different component. Thus, the behavior of each variant can be put in a product by adding or removing one of the components. Since the features are alternative, only one of the components will be present in a product. However, in some cases, depending on the variability granularity, it may be appropriated to put both features in a unique component and add internal variability. This issue will be discussed in variability analysis activity in Section 3.3.

The components identified in this activity will maintain the quality of the services in the product line. Thus, identify components considering quality attributes, e.g., modifiability and reusability, is appropriated. However, some quality attributes, e.g., security and performance, will be responsibility of the service platform selected as well [17].

Existing software components can be considered for integration in this activity to increase reuse. The next section presents the service identification activity, which provides some guidelines and steps to identify service and service orchestration candidates.

3.2 Service Identification

The identification of service candidates is a challenging task of service-oriented computing [18, 19]. In the context of service-oriented product lines, service identification activity is even harder due to concerns with commonalities and variability.

In the service identification activity, a set of service and service orchestration candidates that support the business processes are identified. Thus, as the services are supposed to support the business processes, it is reasonable to identify them from the business process models [3, 5, 20].

This activity starts with an analysis of the business process models. In this analysis, the processes themselves, their sub-processes and business process activities are considered as service or service orchestration candidates, it depends on their granularity. Concurrently, key business entities are identified, and service candidates are created to implement their life cycle methods, e.g., create, delete, update and retrieve [3]. Finally, service candidates are defined to implement utility functionalities that support the services and service orchestrations identified previously, e.g., logging, monitoring and data transformation, when necessary.

We present a top-down approach for service identification, but it does not exclude existing services to be considered for integration during this activity. The service identification activity provides a service portfolio with all the service candidates identified as output. The next section presents the variability analysis activity.

3.3 Variability Analysis

According to [21], variability is the ability to change or customize software systems. Improving variability in a system implies making it easier to do certain kinds of customizations. Moreover, it is possible to anticipate some types of variability and construct a system in such a way that it is prepared for inserting predetermined changes.

At this point, the possible components, and the service and service orchestration candidates of the service-oriented product line have been identified. During the variability analysis activity, it is defined and documented essential architectural decisions about how the variability presented in the feature model and business processes will be implemented within services and components.

The variability analysis activity starts with an analysis of the component and service candidates identified. The similarities and differences among services should be analyzed with the purpose of reduce the number of service candidates. The similarity analysis consists of comparing the functionality of services in order to join similar services that implement fine-grained variability, e.g., variability that can be implemented by changing a class attribute or method. In this case, services will be joined in a single service with internal variability. The same analysis is realized among the component candidates. At this point, the services and components are no longer candidates anymore.

Subsequently, it is analyzed how the variation points will be implemented within the components. Component-Based Development (CBD) can be used as an implementation technique, i.e., each variant is implemented in a different component. Alternatively, well known variability implementation techniques can be used to implement component internal variability, e.g., aspect-oriented programming, conditional compilation, configuration files and design patterns [22]. The same thing occurs with the services. In this case, service orientation can be used as a technique to

implement variability, i.e., each variant can be implemented in a service. It is the way the current service-oriented applications are customized, changing service order or even the participants of service compositions to implement variability. However, depending on the variability granularity it may be insufficient. A variation point can be implemented changing a class attribute, or a class, a method or even an entire component or service. Thus, in some cases it is necessary to introduce service internal variability.

In order to implement service internal variability, i.e., a unique service that can be customized to different purposes, the service interface, in some cases, must reflect the underlying variability the service contains in its components and classes. Thus, conditional compilation and parameterization can be used with the purpose of change service interfaces or modify the service behavior according to specific customer requirements. The use of code transformation tools is used in [17] to implement service interface variability.

Variability analysis activity produces as output a set of architectural decisions regarding variability that will be specified during architecture specification activity, which is presented in the next section.

3.4 Architecture Specification

In the architecture specification activity, the components, services, service orchestrations and their flows will be specified, i.e., the architecture will be specified. In this activity, the models and specification are produced with variability as all the artifacts of core assets development. Architecture specification requires notations with support for variability representation.

Software architectures are complex entities that cannot be represented in a simple one-dimensional fashion [15]. Since there are different stakeholders involved in a project with particular concerns about the system, it is important to represent the architecture upon different views.

During architecture specification, the first step is the definition of component and service interfaces. Subsequently, different architectural views can be produced: structural view, layer view, interaction view, dependency view, concurrency view and physical view. Each view is described in detail next.

The *structural view* represents the architecture static structure. This view shows the components, services and service orchestrations of the architecture. The *layer view* presents the services organized in their layers. The *interaction view* shows how the services and components communicate to realize a specific functionality. The *dependency view* presents dependence information among services and components. The *concurrency view* shows parallel communication among services and components, but it can be represented in the interaction view as well. Finally, the *physical view* shows how the services and components are distributed and the protocol of communication.

Some UML diagrams with stereotypes and variability extensions, such as [23, 24], can be used to create these views. As examples, the component diagram can be used to represent the structural view and dependency view of components, the interaction and concurrency view can be represented with sequence diagrams, and the dependency view of services can be created with interfaces, stereotypes and dependency arrows in class diagrams.

The next section presents a case study on the conference management domain using the approach.

4. Case Study

In this section, we introduce an initial case study on the conference management domain in order to clarify and explain our approach. The case study consists of a service-oriented product line that intends to produce customized service-oriented applications for the management of different conferences.

Part of the feature model of the service-oriented product line is presented in Figure 3, and its features are described next.

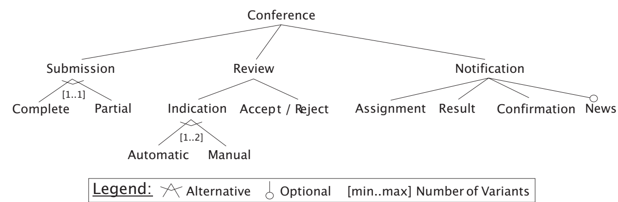

Figure 3. Feature Model.

- Submission: authors can submit their complete papers or, first submit the abstract, followed by the complete version. Complete and partial submissions are alternative features.

- Review: the indication of papers to reviewers can be made automatically and/or manually. Reviewers can also accept or reject paper indications. Automatic and manual indications are not exclusive, they can work together.

- Notification: the system can send information to reviewers about paper assignments. It can send acceptance or rejection (result) information to authors. It can also send event news, e.g., deadlines, and confirmation messages, e.g., paper or review submitted, to authors and reviewers. Event news notification is an

optional feature. Assignments, confirmation and result notifications are mandatory.

Applying the technique for component identification, we finish with the following component candidates: *complete submission*, *partial submission*, *review management*, *automatic indication*, *manual indication*, and *assignment, result, confirmation and news* notification components. The complete and partial submission components were separated because they are alternative features, and only one of them will be bound to an application. The same thing for the automatic and manual indication components, which are an alternative non-exclusive choice, only one, or both will be present in an application. The variants and mandatory sub-features of the notification feature were also put each one in a different component. There are other components, e.g., access control, user management that were excluded from the paper due to space limitations.

Figure 4 shows the simplified paper submission business process. It starts with two optional activities, authors submit the abstract of the paper and receive a confirmation. Afterward, the authors submit the complete version of the paper and receive a confirmation again. Finally, after the reviews finish, the authors receive the result (acceptance and rejection) messages.

Figure 5 shows the simplified review business process. First, the system indicates papers to reviewers automatically and/or manually (chair indication). The reviewers receive the notification about the papers to review. They can reject or accept the reviews, and next, they receive a confirmation about the action they have performed. Finally, the reviewers submit their reviews and receive a confirmation again.

From these business processes, the following service candidates were identified: *abstract submission*, *paper submission*, *review management*, *notification* and orchestration services (*submit process, review process*) for the whole processes. The components of access control and user management mentioned above do not need to be exposed as services because they do not bring any business value to these business processes.

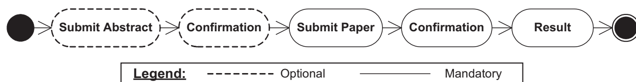

Figure 4. Submission Business Process.

Figure 5. Review Business Process.

After the identification of the components and services candidates, we try to reduce the number of candidates defining how the variability will be implemented. For instance, the abstract submission and paper submission service candidates can be reduced to only one service. However, variability is introduced to the service interface in order to reflect that the service operation submit paper abstract is optional. The automatic and manual indication components, which assign papers to reviewers can be implemented in a unique component, but the variability should be introduced internally using SPL variability techniques, e.g., design patterns. In the case of the notification feature, its sub-features (assignment, results, confirmation and news) were put all in a unique component with internal variability because the variability granularity of these sub-features was low. We also use only one service with internal variability to exposed the notification component, however, this service also required interface variability in order to reflect that the news notification feature is optional.

During architecture specification, the architectural views are created. Figure 6 shows a dependency view of the orchestration service for the submission business process. As it can be seen, the submission service contains a variable operation in order to reflect the variability implemented in the partial and complete submission components. The same thing for the notification service, which has an optional operation as well to reflect that the news feature is optional. As another architectural view example, Figure 7 shows the interaction view of the submission process. The steps related with partial submission (submit abstract and its confirmation message) will be removed of the documentation when the feature complete submission is selected.

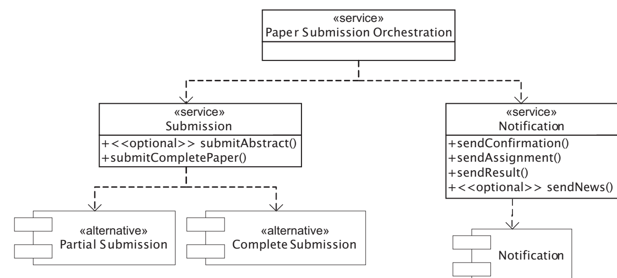

Figure 6. Dependency View.

5. Related Work

Two different approaches for business process modeling based on product line principles exploiting commonalities and variability through domain engineering are presented in [8, 9]. Both works realize processes able to adapt themselves to different customers or market segment needs. Thus, the resulting SOA systems that automate them will be

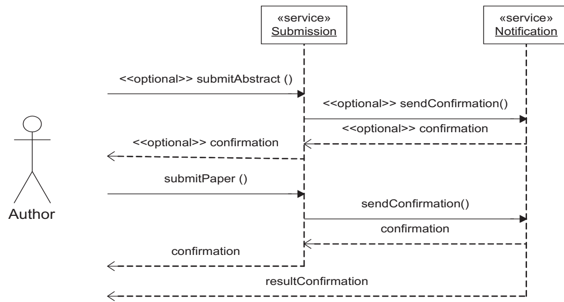

Figure 7. Interaction View.

suitable to different customer needs as the underlying processes. However, none of them concerns the identification of services candidates or gives information about the components that realize the service implementation. In addition, the work is not concerned with architecture specification and documentation, and focus on web service technologies such as BPEL. We solve these gaps in our work providing information on how to identify and specify services and components regarding variability issues. Moreover, our approach does not focus on any specific technology.

An approach for developing service-oriented product lines is presented in [18, 25]. It proposes a service identification method based on the feature binding analysis technique [26]. However, it does not consider the business processes and it may identify service candidates that are not aligned with the business goals. The service identification technique of our approach is based on the techniques used in service-oriented development [3, 20]. Thus, we identify services from an analysis of the business processes.

In [17], a development process for web services is proposed. It analyzes a particular software product line development process and compares it with the service-oriented product line process proposed. It concludes the paper with an example for a service-oriented product line web store that basically uses a code tranformation tool to implement service interface variability. In our work, we suggest some techniques for the implementation of service interface variability, not only the use of code transformation tools, but also well known techniques used in SPL, e.g., conditional compilation and parametrization.

6. Conclusions and Future Work

This work presents a contribution to the combination of SOA and SPL concepts. In particular, how these concepts can be used together to achieve desired benefits such as improved reuse, decreased development costs and time to market, and production of flexible applications customized to specific customers or market segment needs.

In order to achieve these goals, we presented an approach for service-oriented product line architectures that introduces the concepts of managed variability into service-oriented world and uses a two life-cycle model as in SPL engineering, however, only core assets development is considered in this work. These concepts were introduced in order to provide support for high customization and systematic planned reuse during service-oriented development. In this context, services are developed to be reused in specific contexts and service-oriented applications can be developed rapidly and customized according to specific customer requirements. We also present a case study on the conference management domain clarifying and explaining the activities of the approach.

As a future work, we are planning to apply this service-oriented product line architecture approach to others domains and validate the real benefits of the combination of SOA and SPL that we have used in this work. In addition, we are performing a case study using different technologies and techniques for service internal variability implementation in order to identify the real differences, if they exists, from object-oriented and component-based variability implementation.

Acknowledgements

This work was partially supported by the National Institute of Science and Technology for Software Engineering (INES[1]), funded by CNPq and FACEPE, grants 573964/2008-4 and APQ-1037-1.03/08 and Brazilian Agency (CNPq process number 475743/2007-5).

References

[1] F. J. v. d. Linden, K. Schmid, and E. Rommes, *Software Product Lines in Action: The Best Industrial Practice in Product Line Engineering*. Secaucus, NJ, USA: Springer-Verlag New York, Inc., 2007.

[2] S. Carter, *The New Language of Business: SOA & Web 2.0*. IBM Press, 2007.

[3] A. Arsanjani, S. Ghosh, A. Allam, T. Abdollah, S. Ganapathy, and K. Holley, “SOMA: A method for developing service-oriented solutions,” *IBM System Journal*, vol. 47, no. 3, pp. 377–396, 2008.

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

[5] T. Erl, *Service-Oriented Architecture: Concepts, Technology, and Design*. Upper Saddle River, NJ, USA: Prentice Hall, 2005.

[1] INES - http://www.ines.org.br

[6] M. P. Papazoglou and W.-J. V. D. Heuvel, "Service-oriented design and development methodology," *International Journal of Web Engineering and Technology (IJWET)*, vol. 2, no. 4, pp. 412–442, 2006.

[7] A. Helferich, G. Herzwurm, and S. Jesse, "Software product lines and service-oriented architecture: A systematic comparison of two concepts," in *SPLC '07: 11th International Software Product Line Conference*, IEEE Computer Society, 2007.

[8] N. Boffoli, D. Caivano, D. Castelluccia, F. M. Maggi, and G. Visaggio, "Business process lines to develop service-oriented architectures through the software product lines paradigm," in *SPLC '08: 12th International Software Product Line Conference*, pp. 143–147, 2008.

[9] E. Ye, M. Moon, Y. Kim, and K. Yeom, "An approach to designing service-oriented product-line architecture for business process families," in *ICACT '07: 9th International conference on Advanced Computing Technologies*, pp. 999–1002, 2007.

[10] S. Cohen and R. Krut, eds., *Proceedings of the First Workshop on Service-Oriented Architectures and Software Product Lines, 11th International Software Product Line Conference*, 2007.

[11] K. Pohl, G. Böckle, and F. J. van der Linden, *Software Product Line Engineering: Foundations, Principles and Techniques*. Secaucus, NJ, USA: Springer-Verlag New York, Inc., 2005.

[12] L. Northrop, "Sei's software product line tenets," *IEEE Software*, vol. 19, pp. 32–40, July 2002.

[13] A. Arsanjani, "Service-oriented modeling and architecture," tech. rep., Service-Oriented Architecture and Web services Center of Excellence, IBM, 2004.

[14] A. Arsanjani, L.-J. Zhang, M. Ellis, A. Allam, and K. Channabasavaiah, "S3: A service-oriented reference architecture," *IT Professional*, vol. 9, no. 3, pp. 10–17, 2007.

[15] L. Bass, P. Clements, and R. Kazman, *Software Architecture in Practices*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 2003.

[16] C. Szyperski, "Component technology: what, where, and how?," in *ICSE '03: 25th International Conference on Software Engineering*, pp. 684–693, IEEE Computer Society, 2003.

[17] S. Günther and T. Berger, "Service-oriented product lines: Towards a development process and feature management model for web services," in *SPLC '08: 12th International Software Product Line Conference*, pp. 131–136, 2008.

[18] J. Lee, D. Muthig, and M. Naab, "An approach for developing service oriented product lines," in *SPLC '08: 12th International Software Product Line Conference*, pp. 275–284, IEEE Computer Society, 2008.

[19] D. Kang, C. yang Song, and D.-K. Baik, "A method of service identification for product line," in *ICCIT '08: 3rd International Conference on Convergence and Hybrid Information Technology*, vol. 2, pp. 1040–1045, 2008.

[20] A. Erradi, S. Anand, and N. Kulkarni, "Soaf: An architectural framework for service definition and realization," in *SCC '06: Proceedings of the IEEE International Conference on Services Computing*, pp. 151–158, IEEE Computer Society, 2006.

[21] J. V. Gurp, J. Bosch, and M. Svahnberg, "On the notion of variability in software product lines," in *WICSA '01: 2nd Working IEEE/IFIP Conference on Software Architecture*, p. 45, 2001.

[22] C. Gacek and M. Anastasopoules, "Implementing product line variabilities," *SSR '01: Symposium on Software Reusability*, vol. 26, no. 3, pp. 109–117, 2001.

[23] H. Gomaa, *Designing Software Product Lines with UML: From Use Cases to Pattern-Based Software Architectures*. Addison Wesley, 2004.

[24] M. Razavian and R. Khosravi, "Modeling variability in business process models using uml," in *ITNG '08: 5th International Conference on Information Technology - New Generations*, pp. 82–87, 2008.

[25] J. Lee, D. Muthig, and M. Naab, "Identifying and specifying reusable services of service centric systems through product line technology," in *SPLC '07: 11th International Software Product Line Conference*, IEEE Computer Society, 2007.

[26] J. Lee and K. C. Kang, "Feature binding analysis for product line component development," in *PFE '03: 5th International Workshop on Software Product-Family Engineering*, pp. 250–260, 2003.