Variability in Service-Oriented Systems: An Analysis of Existing Approaches

Holger Eichelberger, Christian Kröher, and Klaus Schmid

Software Systems Engineering, University of Hildesheim
Marienburger Platz 22, 31141 Hildesheim, Germany
{eichelberger,kroeher,schmid}@sse.uni-hildesheim.de

Abstract. In service-oriented systems services can be easily reused and shared without modification. However, there are business situations where a variation of services is needed to meet the requirements of a specific customer or context. Variation of software systems has been well researched in product line engineering in terms of Variability Implementation Techniques (VITs). While most VITs focus on the customization of traditional software systems, several VITs have been developed for service-oriented systems. In this paper, we discuss the problem of service customization and provide an overview of different VITs for service variability. For this purpose, we will define four dimensions to describe, characterize and analyze existing VITs: the *technical core idea*, the *object of variation*, the *forms of variation*, and the *binding time*.

1 Introduction

Customization of software systems is current practice in industry to meet the requirements of customers in a qualitative and timely manner. The most frequent reasons for customization are: novel functionality [10], optimization for quality of service aspects [8], and seamless integration into existing infrastructures [5]. Companies face ever-increasing demands on customization due to growing numbers of requirements and rising complexity of software systems.

In Service-oriented Computing (SoC) a typical approach to satisfy varying requirements is to add, remove, or replace services. However, there are situations where the customization of existing services is needed. This could be realized by implementing a completely new variant of a service, but it is more appropriate from a business point of view to customize the service implementation as needed. This will lower the development effort and increase reusability. However, SoC does not provide for the customization of services in terms of tailoring individual aspects of a single service.

An industry best practice to achieve tailor-made systems with low effort and high quality are Software Product Line Engineering (SPLE) methods [9]. The key idea of SPLE with respect to customization is to focus on the differences (called *variabilities*) among similar systems instead of repeating the development. A *variability model* represents all variabilities on an abstract level (including constraints among them) and is used to derive a valid product configuration for instantiation. A *Variability Implementation Technique* (VIT) is an approach to realize variability according to a

C. Liu et al. (Eds.): ICSOC 2012, LNCS 7636, pp. 516–524, 2012.
© Springer-Verlag Berlin Heidelberg 2012

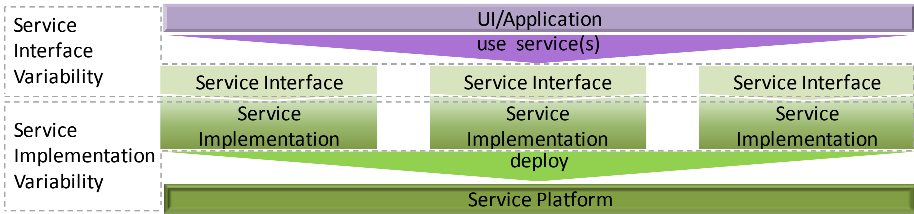

Fig. 1. The role of variability in services

configuration. As the problem of customization is also relevant for SoC, there is a need for approaches that integrate VITs with service-oriented technologies.

In this paper, we will provide a classification of VITs for SoC. The scope of this paper is on service variability including variability of interfaces and implementations. Other forms, like variabilities in business processes, service compositions, or service platforms are out of scope due to space restrictions.

The paper is organized as follows: in the next section, we detail the dimensions to analyze and characterize VITs. Section 3 will describe the approach of the literature study we carried out to identify VITs. The results of the analysis will be presented in Section 4. In Section 5, we will draw conclusions and point out future challenges.

2 Characterizing Variability Implementation

The combination of services with VITs forms the problem space of this work. In this section, we identify dimensions for analyzing and characterizing VITs for SoC. We will use these dimensions to classify the results of our analysis in Section 4 as they provide a good basis for selecting a specific variability technique in practice.

D1. ***Technical core idea***: The VITs described in literature are at the heart of our analysis. For each VIT we will discuss the technical core idea, the individual prerequisites, and the provided capabilities.

D2. ***Variability object***: We differentiate between service interface and service implementation variability as shown in Fig. 1 due to the scope of this paper. *Service interface variability* allows customizing the interface of a service. Typically, this also requires *service implementation variability*. Service implementation variability enables the customization of the implementation of a service (and thus its behavior).

D3. ***Form of the variation***: *Optional*, *alternative*, and *multiple selection* are well known forms of variation in SPLE [9]. *Extension* as a form of variation is particularly relevant to SoC as it supports variability without a predefined range of possible variations. Further functionality may extend an existing service (without creating a new service). Extensions are unknown at development time but may be introduced later (this is a typical open-world scenario). Further forms are mentioned in SPLE literature, but as we could not identify these as part of our analysis, we will not list these here.

D4. ***Binding time***: The binding time determines when a decision is made about a variability. This may be either made once and cannot be altered afterwards (*permanent*) or rebinding for a new variation is possible (*volatile*). Different binding

times are discussed in literature [14]. However, we will focus on a representative set for SoC. At *compile time*, variability binding is performed during the build process by mechanisms such as pre-processors. At *initialization time*, variability binding happens during the startup phase, e.g., based on a configuration file. *Run-time* binding subsumes all cases of binding variability during the execution of a service.

3 Literature Study

We performed a literature study in order to systematically survey existing VITs and to classify them according to the dimensions introduced in Section 2. We defined a strategy based on the guidelines for systematic reviews by Kitchenham and Charters [7] in order to structure our survey. However, our goal is not an evidence-based analysis, but to ensure completeness and correctness of the identified literature. In this section, we will briefly describe our strategy.

We performed our literature search using the most prominent search engines as publication sources[1]. For the search queries, we used six different search strings (combinations of service or SOA and variability, product-line, or SPLE) to cover the entire range of available literature on that topic. We used these search strings with each search engine. The individual searches yielded more than two thousand papers from which we selected about two hundred to be relevant, that discuss variability in SoC (based on reading the title and the abstract). After eliminating duplicates, we applied a set of inclusion and exclusion criteria (topicality and maturity of the approach, focus of the approach, etc.) yielding the final set of relevant papers.

As a result, the literature study revealed 20 VITs in total, which are in principle relevant to service-oriented computing. However, due to space restrictions we decided to focus in this paper exclusively on service variability as discussed in Section 2.

4 Analysis of Variability Implementation Techniques

In this section, we will present the results of our analysis. Please note, that we introduce descriptive names for the VITs for clear identification and ease of reading. The next sections below follow the sequence of dimensions defined in Section 2.

4.1 Technical Core Ideas

A VIT describes a specific way of realizing variability. In this section, we introduce the identified VITs, their technical core idea, and the addressed service technologies. These are summarized in Table 1[2].

[1] ACM Digital Library: `http://dl.acm.org/`, IEEE Computer Society: `http://www.computer.org`, Google Scholar: `http://scholar.google.com/`, and Citeseer: `http://citeseerx.ist.psu.edu/`

[2] Realization approaches and SOAP/WSDL are derived from the VITs; OSGi and REST are representative examples of service technologies (marked as optional if concluded to be applicable).

Table 1. Realization approaches and service technologies required by VITs

m: mandatory, o: optional		PP	CbSi	FOPbR	CW	ASW
Realization Approach	Component-based	-	m	-	-	-
	Aspect-oriented	-	o	-	-	m
	Interception	-	-	-	-	m
	Feature-oriented	-	o	m	m	-
	Generative	m	-	-	-	-
Service Technology	OSGi	o	o	o	o	-
	SOAP / WSDL	o	o	m	o	m
	REST	o	o	o	o	-

The *Pattern Plugin* (*PP*) approach [12] is a generative approach, i.e. service variants are generated from a variant-enabled design model. The design model is an extension of UML, and includes common and variable parts (as variation points) of a Service-Oriented Architecture (SOA). A variant is expressed as a stereotyped model element (variation model) which holds the information on the actual variation. A SOA variant is defined by selecting appropriate variants. The variation models of the variants are composed into the primary design model via pattern plugins. A pattern plugin describes an individual variant. The composed model can finally be transformed into code artifacts. The encapsulation of variability in variation models and related plugins allows to arbitrarily selecting the service technology (marked as optional in Table 1).

Component-based Service Implementation (*CbSI*) [10] adds a component layer as a refinement of services and realizes variability on the component level. The approach is rather generic and more a conceptual framework than a single approach. For example, a service can be implemented as an optional component (service implementation). Other VITs like aspects, features, etc. are possible (optional in Table 1). *CbSI* provides variability of the implementation, while the service layer is variation-free (with respect to service implementation variability). Thus, this VIT does not require a specific service technology (optional in Table 1).

The *FOP-based Refinement* (*FOPbR*) approach [1] relies on Feature-Oriented Programming (FOP). A feature represents an increment in functionality, which affects one or multiple services simultaneously. *FOPbR* encapsulates the code of a feature into a feature module. A feature module consists of a set of refinements for a service's base code which are enacted by joining the base and the feature code. As a prerequisite, FOP needs to be available for the implementation language, such as for Java [4] or WSDL [2] (mandatory in Table 1). The use of other service technologies is unclear, but we expect this to be optional.

The *Class Wrapper* (*CW*) approach [13] also applies FOP techniques to SoC. In contrast, *CW* uses Java HotSwap to update bytecode in place using the same class identity. HotSwap is required to add features in terms of base classes and wrappers to the service implementation (plain Java code). Base class code updates only internal algorithms without affecting the class schema. Wrappers are used to introduce new elements such as additional methods. In order to invoke the functionality provided by the wrapper, HotSwap is used to update all object references of the changed class.

Table 2. Variability objects addressed by VITs

x: supported		PP	CbSI	FOPbR	CW	ASW
Variability Objects	Service Interface	x	-	x	-	-
	Service Implementation	x	x	x	x	x

While *CW* is conceptually similar to *FOPbR*, however, it would also allow the volatile rebinding at runtime as we will discuss in Section 4.4. The customization of arbitrary Java code yields service technology-independence (optional in Table 1).

The *Aspect Service Weaver* (*ASW*) approach [11] relies on Aspect-Oriented Programming (AOP) and message interception. *ASW* intercepts existing service message chains (based on SOAP) between service consumer and provider. If a message includes a request for a method that the service does not support, advice services are required. An advice service implements additional code (the variability) that can be woven into existing services. The *ASW* tool [3] supports this for SOAP and Web services (marked as mandatory in Table 1).

4.2 Variability Objects

A variability object is an element of a SoC that is supposed to vary. As discussed in Section 1 we restrict our scope to services, i.e. services interface variability and service implementation variability. In this section, we describe which variability objects can be supported by which VIT (cf. Table 2) and how variation is realized.

The *PP* approach supports interface and implementation variability. The basic service, which is supposed to vary, is described by a service operation description and it's in- and outputs. Each variant is given as a variation model. In case of an interface variant, the model specifies the modified interface, the affected in- and outputs as well as a variant description. For an implementation variant, the model lists the affected operations, in- and outputs. The variation models are associated with the basic service model. Given a specific selection of the variants for a basic service, a code generator produces the service interfaces and the related service implementation variants.

In *CbSI*, implementation variability is enabled by the component layer. Each service implementation is realized by at least on component. The selection of the components for the implementation makes up the variability. Thus, the same service may provide different functionality based on the selected components. However, there is no mechanism that ensures that the in- and outputs of the service interface (service layer) and the service implementation (component layer) match. This must be done on a more abstract level, e.g. in the variability model which controls the customization.

FOPbR supports both, interface and implementation variability. An interface variant is a refinement of a WSDL interface definition [2] that includes the affected service methods. An implementation variant is realized as a class refinement introducing new and/or modified functionality. The set of related interface and class refinements represents a feature, which can be applied to the service's base implementation.

The *CW* approach only supports implementation variability. The base program is given as plain Java. Each feature consists of a set of classes and wrappers. A class

Table 3. Forms of variation supported by VITs

x: supported		PP	CbSI	FOPbR	CW	ASW
Form of Variation	Optional	x	x	x	x	x
	Alternative	x	x	x	x	x
	Multiple Selection	-	x	(x)	(x)	(x)
	Extension	-	-	(x)	(x)	(x)

may introduce new functionality, while a wrapper refines one of the base classes in terms of altered methods. The wrapper class therefore holds an object of the wrappee class, which enables the wrapper to call the basic methods of the base class first and then manipulate the results by calling additional methods introduced by the wrapper.

The *ASW* only supports implementation variability. A functional variant, e.g. a specific method, is encapsulated as an advice service. If this functionality is requested by a service call, the *ASW* weaves the code of the advice service into the base service. Joinpoints identify the functionality which should be modified in the service base code [6]. The advice service can then be woven before, after or around this joinpoint.

4.3 Forms of Variation

Form of variation describes how specific variants can be selected. In this section, we discuss the support of the VITs for the forms of variation (cf. Table 3).

The *PP* approach supports optional and alternative forms of variation. Typically, each variant provides functionality describing a service implementation or a service interface variant. Thus, the selection is either optional or an alternative, but there is no support for selecting multiple variants or the explicit modeling of extensions.

In *CbSI* a component may be optional, an alternative, or combined with other components (multiple selection) to implement a service. However, components cannot be added after development time (extension) as the components are linked to specific services in the service layer and later (re-)linking of components is not supported.

The other VITs support optional, alternative, and multiple selection as well as extension in principle. In *FOPbR* and *CW* the use of refinements or wrappers is optional. Multiple refinements or wrappers which affect the same functionality of a service will override previously applied variants. Extensions to the base implementation after development time can be applied by refinements and wrappers. As both VITs need access to the service code, we put extension in Table 3 in brackets. Similar for multiple selection as it is not directly supported by the technique, but can be simulated.

In *ASW* an advice service may or may not be woven into an existing service (optional). It may also be possible to select one or multiple advice services as long as the advice services will not affect the same joinpoint (cf. Section 4.2). This will also result in overriding previously applied variants as in *FOPbR* and *CW*. Introducing new functionality, which was unknown at development time, requires the joinpoints of a service to be accessible. As again some support for extension is given, but no full support we put this in brackets in Table 3. Similar for multiple selection.

Table 4. Binding times supported by VITs

p: permanent, v: volatile		PP	CbSI	FOPbR	CW	ASW
Binding Time	Compile Time	p	p	p	-	(v)
	Initialization Time	-	-	-	(v)	(v)
	Runtime	-	-	-	v	v

4.4 Binding Times

The binding time defines when a decision for a specific variant must be made. In this section, we describe the binding times supported by the individual VITs (cf. Table 4).

PP, *CbSI*, and *FOPbR* only support permanent compile time binding. In *PP*, customization is realized by replacing existing or adding additional variants. This must be done before the generation process and, thus, at the latest at compile time. Replacing variants in the design model after the generation will not affect the generated code (permanent binding). In *CbSI*, the components of a service implementation are instantiated and composed at compilation time. In *FOPbR* the features are handled by the compiler which applies them to the corresponding base code. Thus, the selection of variants must be done at compile time and cannot be changed afterwards.

CW supports volatile runtime binding via Java HotSwap which enables class (re-) binding. While the authors do not explicitly propose to use this approach at initialization time, this is, however, also possible (marked with brackets in Table 4).

Typically, AOP approaches are capable of compile time binding through static weaving and (some form of) runtime binding by dynamic weaving [6]. *ASW* explicitly supports volatile binding at runtime but may also be applied at initialization or compilation time (again marked with brackets in Table 4). Further, *ASW* allows reweaving code of advice services (volatile binding).

5 Conclusion

Customization of SoC is typically done by adding, removing or exchanging services. However, there are situations where variations of the characteristics of services are needed. We presented an overview of existing VITs for services and characterized them with respect to core idea, variability object, form of variation, and binding time.

In our analysis, we also identified gaps and challenges. The characterized VITs support only WSDL-based web services explicitly. There is no explicit proof-of-concept for other service technologies like OSGi or REST. While the variability objects are well supported, none of the VITs provides guidance to ensure that modifications to service interfaces also match the related implementation. As the modifications are also local to a service, there is no guarantee that the interfaces on caller as well as on callee side are customized. Regarding the form of variation, the VITs do not support the open-world scenario, i.e. extension of existing services with functionality which was unknown at development time (unless the code is accessible). Further, all binding times are (partially) supported but only one VIT supports all binding times. Ideally, customization should be possible to perform at all binding times.

The most obvious result of our analysis is that no VIT supports all dimensions in a comprehensive manner. Each approach focuses on a subset of elements of the dimensions and, thus, provides specific mechanisms for these elements. However, in SoC, we need integrated solutions that support all aspects of service variability appropriately. An integrated solution will enable the customization of service (and SoC in general) across technology and business boundaries with low effort and high quality.

In future work, we will focus on such an integrated VIT for SoC. For this purpose, we will consider already analyzed VITs for variability objects such as service platforms, service deployment, service composition, and business processes.

Acknowledgments. This work is partially supported by the INDENICA project, funded by the European Commission grant 257483, area Internet of Services, Software & Virtualisation (ICT-2009.1.2) in the 7th framework programme.

References

1. Apel, S., Kaestner, C., Lengauer, C.: Research Challenges in the Tension Between Features and Services. In: 2nd Intern. Workshop on System Development in SOA Environments, pp. 53–58 (2008)
2. Apel, S., Lengauer, C.: Superimposition: A Language-Independent Approach to Software Composition. In: Pautasso, C., Tanter, É. (eds.) SC 2008. LNCS, vol. 4954, pp. 20–35. Springer, Heidelberg (2008)
3. Baligand, F., Monfort, V.: A Concrete Solution for Web Services Adaptability Using Policies and Aspects. In: 2nd Intern. Conference on Service Oriented Computing, pp. 134–142 (2004)
4. Batory, D., Sarvela, J.N., Rauschmayer, A.: Scaling Step-Wise Refinement. In: 25th Intern. Conference on Software Engineering, pp. 187–197 (2003)
5. Istoan, P., Nain, G., Perrouin, G., Jézéquel, J.-M.: Dynamic Software Product Lines for Service-Based Systems. In: 9th Intern. Conference on Computer and Information Technology, vol. 2, pp. 193–198 (2009)
6. Kiczales, G., Lamping, J., Mendhekar, A., Maeda, C., Lopes, C., Loingtier, J.-M., Irwin, J.: Aspect-Oriented Programming. In: Aksit, M., Auletta, V. (eds.) ECOOP 1997. LNCS, vol. 1241, pp. 220–242. Springer, Heidelberg (1997)
7. Kitchenham, B., Charters, S.: Guidelines for Performing Systematic Literature Reviews in Software Engineering. Technical Report EBSE-2007-01, School of Computer Science and Mathematics Keele University, Staffs ST5 5BG, UK (2007)
8. Li, Y., Zhang, X., Yin, Y., Wu, J.: QoS-Driven Dynamic Reconfiguration of the SOA-Based Software. In: Intern. Conference on Service Sciences, pp. 99–104 (2010)
9. van der Linden, F., Schmid, K., Rommes, E.: Software Product Lines in Action - The Best Industrial Practice in Product Line Engineering. Springer (2007)
10. Medeiros, F.M., de Almeida, E.S., Meira, S.R.L.: Towards an Approach for Service-Oriented Product Line Architectures. In: 3rd Workshop on Service-Oriented Architectures and Software Product Lines (2009)
11. Monfort, V., Hammoudi, S.: Towards Adaptable SOA: Model Driven Development, Context and Aspect. In: Baresi, L., Chi, C.-H., Suzuki, J. (eds.) ICSOC-ServiceWave 2009. LNCS, vol. 5900, pp. 175–189. Springer, Heidelberg (2009)

12. Narendra, N.C., Ponnalagu, K., Srivastava, B., Banavar, G.S.: Variation-Oriented Engineering (VOE): Enhancing Reusability of SOA-Based Solutions. In: 5th IEEE Intern. Conference on Services Computing, pp. 257–264 (2008)
13. Siegmund, N., Pukall, M., Soffner, M., Köppen, V., Saake, G.: Using Software Product Lines for Runtime Interoperability. In: Workshop on Reflection, AOP and Meta-Data for Software Evolution, pp. 1–7 (2009)
14. Svahnberg, M., van Gurp, J., Bosch, J.: A Taxonomy of Variability Realization Techniques. Software – Practice and Experience 35(8), 705–754 (2005)