Engineering Virtual Domain-Specific Service Platforms

Specific Targeted Research Project: FP7-ICT-2009-5 / 257483

Variability Implementation Techniques for Platforms and Services (Interim)

Abstract

Creating domain-specific service platforms requires the capability of (automatically) customizing and configuring service platforms according to the specific needs of a domain. In this deliverable we address this demand. We focus on how to create customized service platforms using variability implementation techniques.

The focus is on understanding variability implementation in the context of a service platform specific situation and with respect to the specific demands of the INDENICA project. Towards this end, we provide an analysis of this situation, structure and analyze a large body of relevant approaches for customizing service technologies and finally describe the core concepts that provide the basis of the INDENICA approach to implement the customization of service platforms.

Document ID:	INDENICA – D2.1.1
Deliverable Number:	D2.1.1
Work Package:	WP2
Type:	Deliverable
Dissemination Level:	PU
Status:	final
Version:	1.0
Date:	2011-09-30
Author(s):	SUH, SAP, SIE, TEL

Project Start Date: October 1st2010, Duration: 36months

Version History

0.1	01. July 2011	initial version
0.7	7. October 2011	Release version
1.0	14. October 2011	Final version

Document Properties

The spell checking language for this document is set to UK English.

Table of Contents

Table of Figures

1 Introduction

The main focus of work package 2 within the INDENICA project is the customization of service platforms. As part of this effort, this deliverable addresses variability implementation, in particular variability implementation techniques in service-based systems in general and demands and concepts for variability implementation in INDENICA in particular. This is of course related to the modelling of variability, which will be addressed in deliverable D2.1. For the purpose of this deliverable we will take as a basis variability modelling techniques that are quite common in literature like feature modelling and decision modelling.

In this deliverable mainly the initial concepts of variability implementation are in focus. This will be further developed and extended by deliverable D2.1.2, which will cover all concepts that are developed for variability implementation as part of this project. As a consequence, this deliverable plays more a preparatory role within the project.

In Section 2, we introduce the main concepts of variability implementation, as these provide a basis for understanding the remainder of the deliverable. This chapter is still rather generic, as it mainly relies on the general work in product line engineering, in order to derive the conceptual basis.

Section 3 then discusses relevant work in the area of variability implementation in service-oriented systems. (We extended the scope to variability implementation techniques in middleware, as there is actually very little work, particularly on specific techniques for service platforms and we deemed the general work also relevant to this deliverable.) We approached this work by defining a general template that covers the main aspects of service implementation techniques in the form of a pattern (problem-solution pair). The various techniques were identified using a systematic literature review technique as well as input from the industrial partners was sought. The taxonomy, we developed, covers variability implementation techniques for service platforms as they are addressed in the INDENICA context. Using our approach, we could identify nearly two dozen techniques for service variability implementation. We provide an overview of these techniques in Section 3.2. An individual description of the implementation patterns is provided in the appendix.

In Section 4, we define the requirements for variability implementation in service platforms as they are relevant to the INDENICA project. This information is derived from multiple sources, including gathering of feedback from the various industrial partners in the project.

Overall, INDENICA provides rather demanding requirements for variability implementation. In particular, we require - in order to realize the INDENICA vision - a rather general approach to managing variability implementation. However, at this point existing approaches are very specific in terms of their properties (e.g., types of artefacts handled, binding time supported, etc. Thus, in Section 5, we discuss the

main concepts we developed for a more general approach to variability implementation.

2 Basic Concepts in Variability Implementation

The purpose of this section is to provide an overview of major concepts that are relevant to understanding variability implementation. Variability (and hence variability implementation) is a classical product line concept. Thus, we will provide first a short introduction to basic concepts in product line engineering in Section 2.1.

In Section 2.2, we will focus on and explain core concepts in variability implementation. This will provide the conceptual basis for the remainder of this deliverable. Section 2.3 will discuss different types of variability implementation approaches, in order to provide a better understanding of the conceptual landscape, while Section 2.4 will discuss concepts that are particularly relevant to the approach that we will present in this deliverable. Finally, Section 2.5 will discuss the relation between this deliverable and other deliverables in the INDENICA project.

2.1 Software Product Line Fundamentals

In this section, we give an overview on some fundamental terminology in software product line development.[1] First, we discuss how product line engineering differs from traditional software development and how reuse is addressed in product line engineering.

Traditional software development focuses on the development of individual products, typically in a project-based approach, where each project commands its own infrastructure, including all relevant software. In contrast, software product line development addresses the integrated development of a range of similar, but also different products. The products in a product line may support different, individual customers or may address entirely different market segments. Instead of understanding each individual system by itself, software product line engineering looks at the product line as a whole and systematically addresses the differences of the products. Thus, while systems in a product line differ in some characteristics (often referenced as features) to serve the needs of different customers or market segments, the individual systems need also to share a set of common functionality so they can be effectively developed together as a product line.

The distinction between software development for reuse and development with reuse is fundamental in software product line engineering. Development for reuse (domain engineering) provides a basis for the development of individual products in terms of assets designed and realized for reuse. In contrast, development with reuse (application engineering) builds the final products on top of the assets developed for reuse in domain engineering. This distinction is shown as two-life-cycle model in Figure 1. The two-life-cycle model consists of a specific software development lifecycle for domain engineering activities and a second lifecycle for application engineering. The latter relies on the reusable assets created in domain engineering.

[1] This section is partially based on material from [47].

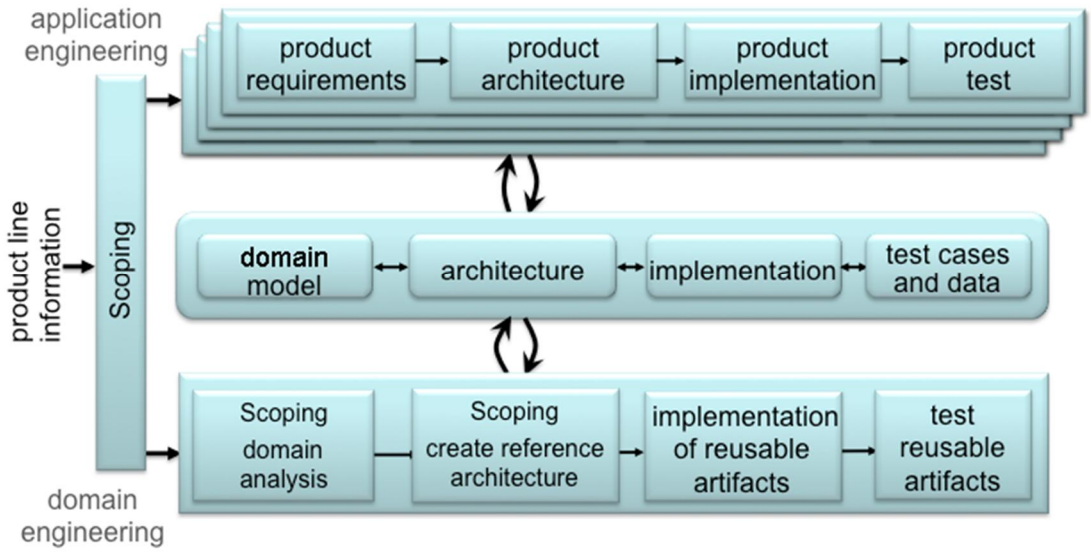

Figure 1: The two-lifecycle model of software product line engineering.

Domain engineering provides a basis for the development of individual products. This basis, the so-called product line infrastructure (also known as core asset base or product line platform[2] [67]) consists of all assets being relevant to the software development during the whole software development lifecycle. To serve for all products that may be developed in the product line, the assets in the product line infrastructure may contain explicit variability. Variation points in the assets describe the potential locations of impact for individual variabilities. For example, in a software architecture model individual services may be tagged as variability points so that the concrete architecture model for a product contains only selected services. This shows also that the core approach in product line engineering to dealing with reuse relies on configuring individual variations.

Application engineering builds the final products based on the product line infrastructure, which usually contains most of the required functionality. The development of a new product based on an existing product line infrastructure consists of eliciting requirements, categorizing requirements as being part of the product line or product-specific and configuring the variabilities in the product line infrastructure, i.e. deriving instantiated versions of the assets that exactly adhere to the requirements of the specific product.

Explicit variability in the assets of the product line infrastructure is a key concept in supporting different customers or market segments. Variability management encompasses all activities for systematically addressing the variability throughout software product line engineering, e.g. defining, representing, implementing or evolving variabilities. In product line engineering, we distinguish three main types of requirements:

[2]The term (product line) platform has a significantly different meaning than in the area of service-based systems. In particular, it differs from (virtual) service platform as defined in D1.1.

- Commonalities are (functional or non-functional) characteristics that are common to all products in the product line and implemented as a part of the product line infrastructure.
- Variabilities are characteristics that may be common to some products, but not to all. Variabilities must be explicitly modelled, implemented and instantiated in a way that allows having it in selected products only.
- Product-specific characteristics are part of only one product. Typically, product-specific characteristics arise in order to address the concerns of individual customers or markets. This type of variability will not be realized directly by the product line infrastructure, but the infrastructure must be able to support this type of extension.

While commonalities and variabilities are provided in domain engineering, product-specific characteristics are exclusively handled in application engineering.

Mastering a range of products instead of individual products is not just a technical topic, e.g. how to effectively realize different functionality among similar but also different products. In addition to architectural and technical concerns, successful software product line development also needs to address business, process and organization concerns [47]. Briefly summarized, specific product line cost models and the integration of technical and marketing-oriented product line planning (product line scoping) are used to address the business concerns. The two-lifecycle model and the differentiation into domain and application engineering are means to approach process concerns. Finally, the adoption of the product line approach may lead to necessary adjustments to the organizational structure as well as to the introduction of specific roles or responsibilities.

2.2 Variability Implementation in the Product Line LifeCycle

Variability management covers the whole software development lifecycle. It starts with the early steps of product line scoping, covers all the lifecycle activities to implementation and testing and finally needs to be considered during evolution. In this section we relate the activity of variability implementation to the software product line lifecycle and highlight particularly the influence of domain and application engineering on variability binding and implementation.

The basis for relating the activity of variability implementation to both lifecycles of the two-lifecycle model shown in Error! Reference source not found. is a clear understanding of concerns. Therefore, we briefly introduce the notion of spaces to separate different concerns regarding variability in software product line engineering in general, and to describe in particular the relation of variability implementation to these spaces.

The Variability Space comprises all issues concerning variability modeling in software product line development. This is, defining and maintaining the product line's variability model, selecting variants for the variation points and the resulting model configuration itself. To be more precise, the variability space covers the following activities and artifacts:

- The Variability Model is an abstraction of all common and varying software assets of the respective software product line. The model illustrates all commonalities and variabilities, their relation and the rules and dependencies between them. Consequently, the model specifies all possible product configurations in terms of commonalities and variabilities.
- The Configuration (Process) describes the process of making decisions or selecting variants to achieve one model representing the description of the final product. This model is called Variability Configuration.

The Asset Space comprises all issues concerning software assets in software product line development. This is, developing and managing the different product line assets, instantiating specific assets for use in a specific product and the resulting asset instances itself. In more detail, the asset space covers the following activities and artifacts:

- The Product Line Assets are any assets that are needed to create the products of the product line and may be used to realize commonalities and variabilities. At this point, all assets are generic assets, e.g. there exist wildcards in the code files which have to be replaced by code fragments with respect to a certain variant decision in a product configuration. Of course, these product line assets need not necessarily to be code fragments. This also includes models for model-driven development, libraries, e.g. different JDBC drivers for support for different databases, or text files for generating manuals. The set of all product line assets is called Product Line Infrastructure.
- The Product Assets are a subset of the product line assets, selected to yield in a final product. The variability inside these assets is reduced to a minimum. In case of pre-compile time binding the assets are variation free (we will introduce the definition of different binding times in Section 2.3). Product-specific characteristics may be added to these assets.
- The process of Instantiation aims at creating the product assets in consistency to the related variability configuration. This includes, among other tasks, code substitution, deletion of assets irrelevant to a specific configuration, combination of several input files for generating new files, and replacement of whole components or services.

Variability implementation, as a fundamental part of domain engineering, provides the basis for the activities of application engineering. As a consequence also variability implementation must be prepared in the domain engineering phase, while the results are used in application engineering. In domain engineering, mostly common and variable product line assets are implemented as part of the product line infrastructure, e.g. as parameter values, alternative components or services, alternative code fragments, as code generation rules, etc. Concrete techniques for implementing variabilities in service platforms and services will be discussed in Section 3.2.

In addition, the variability model is defined in domain engineering. This is not an essential part of variability implementation but it serves as a basis for variability

configuration and therefore product line asset instantiation (resolution of implemented variability).

Application engineering is responsible for deriving concrete products from the product line infrastructure. Therefore, a variability configuration is configured in the variability space, which contains the binding of the individual variabilities according to the concrete requirements of the product. While creating the variability configuration dependencies among the different variabilities need to be taken into account, e.g. it may be the case that variability *A* excludes a variability *B* and, thus, predefines the variability binding for variability *B*. The individual product line assets from the product line infrastructure are then instantiated according to the variability configuration, i.e. irrelevant alternatives are removed from the product. Dependent on the concrete variability implementation technique this can be done e.g. in terms of a configuration file, by passing concrete values to a code generator or by just excluding certain assets from the build process.

2.3 Classification of Variabilities

A high-level classification of variability was given in Section 2.1, which differentiated characteristics of a software product line into commonalities, variabilities and product-specifics. Section 2.2 introduced variability and asset space for separating activities and artefacts according to development concerns. In this section we introduce further categories for characterizing how a variability and its implementation may affect assets of the product line infrastructure. The categories will be used in the remainder of this deliverable to characterize variability implementation techniques for service platforms and services in INDENICA. Therefore, we will discuss the completeness of variabilities, how assets are affected from two points of view (the one of the software lifecycle and the one of the asset itself) as well as when variabilities may be bound.

A variability configuration consists of bindings for individual variabilities, i.e. information on how a variability is bound. The completeness of a variability configuration indicates whether a product can be derived from the configuration or whether further configuration steps are required. A variability configuration may be *complete*, i.e. all variabilities are bound and a concrete product can be derived due to that configuration. A configuration may also be *partial*, i.e. certain (required) variability bindings are left open. Partial variability configurations need to be further concretized by subsequent configurations until a complete binding is reached (also called staged configurations [27]). Partial variability bindings are useful e.g. when application engineering is shared among different departments so that partially bound products, which are preconfigured for a certain market segment by one department, are customized for a specific customer in a subsequent department.

From the point of view of the software lifecycle, variabilities may affect assets in different phases, such as

- Requirements phase: Domain- and application-level functional and non-functional requirements.
- Architecture phase: Architecture descriptions, architectural models, available architectural views and architectural styles etc.

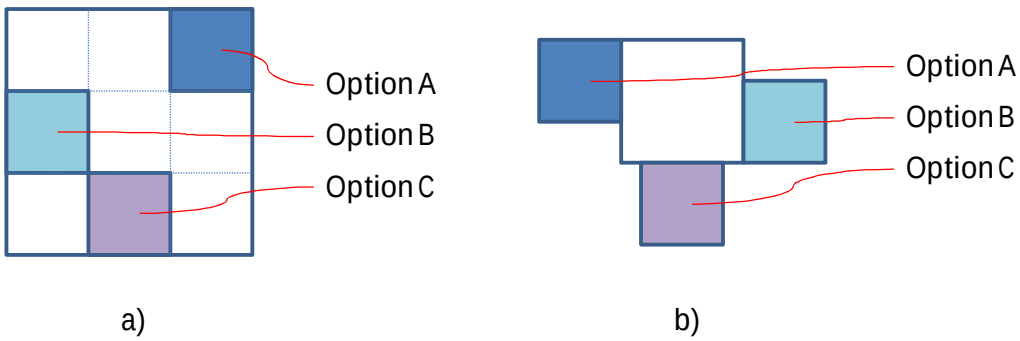

Figure 2: Negative variability vs. positive variability

(based on [37])

- Implementation phase: Source code assets, generation templates, deployment descriptors etc.
- Runtime phase: Active components and services, dependencies and wiring among components, type and granularity of monitoring actions, etc.

Variabilities on levels related to earlier lifecycle phases may impact variabilities in later lifecycle phases in terms of dependencies or constraints. Let us assume that the domain requirements provide a variability whether the system must comply with given real-time constraints. When the real-time variability is selected, certain architectural styles may not be used, i.e. constraints enforcing this may be activated. As a consequence, the code generation may be forced to avoid dynamic bindings wherever possible and at runtime a high-performance server that is able to provide some real-time guarantees is target of the deployment.

If we describe the relation between the variabilities and the commonalities that are relevant to an individual asset, there are two basic forms as outlined e.g. in [37]:

- The reusable asset may contain not only the common, but also all possible variable parts. Then as part of the instantiation parts that are not relevant for a configuration are removed. This is called negative variability or subtractive variability and is illustrated in Figure 2 a). An example of this approach are pre-processor-based approaches as they are often used in the context of the C-programming language.
- The core reusable asset contains only the common parts and possible variable parts are contained in separate assets. In this case the configured asset results from combining the variable parts with the core asset. This is called positive variability or additive variability and is illustrated in Figure 2 b). An example of this is aspect-oriented composition of the assets.

In practice, negative variability is applied more frequently. This is due to the fact that negative variability is supported traditionally using pre-processor statements in C/C++ source code[3].

[3]Pre-processor statements are recognized as a variability implementation technique in product line engineering with similar effects as explained for constants explained in this section.

Traditionally, product line engineering targets the development time of the product, i.e. variability binding and instantiation is typically made during the development of a product. More precisely, the binding of a variable may happen at one of several points in time (binding time) during the software lifecycle such as described in [78]

- Design time – point in time when the architecture of the product is designed.
- Derivation time – defining a product (line) specific architecture from a more general architecture by binding open variation points.
- Check out time - when the assets are obtained from the development repository.
- Compilation time – when the source code assets are processed by the compiler.
- Link time – when the compiled binaries of the product are linked to an executable.
- Startup time – when configuration settings of the product are read, e.g. operation system specific settings, interworking with other installed software, user specific settings, etc.
- Runtime – the variability binding is postponed and affects the executed parts until runtime of the product. While runtime binding is traditionally not in the focus of product line engineering, it recently attracts more interest in dynamic software product lines (DSPL) [39].

The binding time for a concrete variability is usually defined as part of domain engineering and defines the latest point in time when a concrete variability must be bound. Applying different binding times in one product line leads also to partial configurations so that e.g. a subset of the variability is bound at configuration time while others are bound at compile time.

A variability implementation technique is used to realize a certain variability. The implementation technique that is used mostly defines the binding time. Thus, once a specific variability is implemented in a specific way, its binding time is usually fixed. There are few techniques that deal with multiple binding times, as we will discuss in Section 5.2. Being able to vary something that is usually taken as fixed in variability implementation (like the binding time, the implementation technique, etc.) is also called meta-variability [72].

2.4 Variability Implementation Techniques in INDENICA

Several different variability realization techniques can be found in practice and in literature. Some work concentrates on the traditional aspects of product line engineering (excluding runtime variability) such as the realization taxonomies by Muthig and Patzke in [53] or the one by Svahnberg and Bosch in [78]. Recently, work on variability implementation in product line engineering also addressed implementation techniques for runtime variability, including specific variability techniques for service-oriented architectures and service-based systems. We will discuss the current state of the art in detail in Section 3 as a basis for this deliverable. In this section, we briefly summarize the main differences between existing work and variability implementation for service platforms and services in INDENICA.

- Services are typically bound dynamically, i.e. during startup, initialization or at runtime. Work in INDENICA will particularly focus on realizing variability at late binding times as well as for binding time ranges, i.e. for multiple binding times in order to extend flexibility and provide advanced options for runtime adaptivity of services and service platforms.
- Naturally, variability in service-based systems may affect individual services, their functionality, their interfaces or their configuration. This is complemented in INDENICA as the variability realization techniques will particularly influence the service deployment configuration and, thus, support optimized deployment of services in the (virtualized) service platforms.
- Additionally to variability in individual services, INDENICA will provide specific implementation techniques for the variability of service compositions, particularly the variability of service orchestrations.
- A key aspect in INDENICA is the integration of existing platforms into a virtualized service platform, which provides aggregated services of the existing platforms as well as management services for the virtualized platform. While there is work on systematically configuring middleware or even service platforms, variability for a generic virtualized service platform is a specific research challenge in INDENICA.
- Furthermore, work in INDENICA will take quality of service (QoS) aspects in variability models and variation points into account. While work in this field already exists, so far very little work is available at the moment.

2.5 Relation to Other Work in INDENICA

This deliverable discusses variability implementation techniques for service platforms and services to be applied in INDENICA. Techniques for modelling variabilities as a prerequisite for variability binding and instantiation, i.e. the characteristics of a variability, the interrelations and constraints among variabilities as well as special considerations for achieving scalability of variability models will be a topic of D2.1.

Further relationships to other INDENICA deliverables are:

- D1.2.1: Variabilities in the requirements model.
- D3.1: Variabilities in architectural models and in the view-based modelling approach, instantiation of models and variable assets by generative techniques
- D4.1: Configuration of deployment and monitoring.
- D5.2: Concrete variability points in the use cases and industrial platforms.

3 Variability in Service-Based Systems: an Overview

In the previous sections, we focused on variability implementation in general. However, the specific focus of variability in the service world is of particular importance in this deliverable for three reasons:

- INDENICA focuses on the customization and the interoperability of service platforms, thus the customization and hence variability of services is important.
- INDENICA will work with a wide range of different domain-specific service platforms. Thus, we need to be open in terms of the used technologies.
- INDENICA platform customization will need to work at different binding times, depending on the specific contexts in which the customization needs to be done. This requires the availability of a range of different techniques.

In Section 3.1, we will describe a taxonomy that we used to review the existing work in variability of services. Section 3.2 will then discuss the actual variability techniques that we found as part of our overview and their commonalities and differences.

3.1 Taxonomy for Variability in Services and Service Platforms

In this section, we present a taxonomy for classifying and characterizing variability implementation techniques for services. The sources for these techniques are mainly twofold: first, we made an extensive literature survey to capture basically all existing variability implementation techniques that were specifically developed to rely on service-oriented foundations; second, we asked our industry partners to report on techniques that are relevant to them.

As a basis for reporting on the found variability implementation techniques, we used the concept of a pattern. This is an already well-established approach for reporting on software development knowledge. Originally described in the context of design patterns [35], this approach has also been used to describe software architectures [20], or programming approaches(also called idiom) [20]. A major step for using a pattern-based approach is to define the exact structure of the pattern.

The structure that we developed for this deliverable was driven by the goal to have a good index into the problem space that can be used to structure the possible solutions. Thus, on the top-level the structure of our pattern template is rather standard. These are the top-level aspects of the pattern template:

Name: each pattern receives a name. As many patterns we found do not have a name from their authors, we often tried to provide a meaningful name.

Purpose: this is sub-structured and the various facets together describe the situation for which a variability implementation technique is required.

Context: the context provides further restrictions on the situations in which the technique is applicable.

Solution: this describes the solution that is proposed by the pattern.

Further Aspects: any aspects that seem relevant or of interest, but do not fit into the previous categories.

In the following subsections, we will further discuss purpose, context, solution, and further aspects, as these consist of several (non-trivial) entries. In the Appendix A we describe the detailed application of this template on a number of techniques that we could identify.

3.1.1 Purpose

The purpose section of the pattern template consists of four main categories. These are:

Description: this provides a short description of the problem in variability of service-based systems and service platforms that is addressed by the variability implementation technique. The description is a short summary in free text format.

Variability Object: the variability object is the part of the service platform, service, or application that is supposed to vary. This is further explained below.

Form of Variation: in variability modelling we typically differentiate several forms of variability. This can also be used to distinguish between variability implementation problems. Again, we will describe this further below.

Binding Time: finally an important issue is when is it necessary to decide on the specific variability that is relevant in a specific situation. This is also called the binding time. Again, we will further discuss this below.

Variability Object

In general in product line engineering variability implementation typically relates to some fragments of code (lines of codes, methods, components, etc.). Given the specific context of service-oriented computing in general and INDENICA in particular, we can be much more precise. Figure 3 shows a reference architecture, we use throughout this deliverable to identify the various levels on which variability can be relevant to an INDENICA platform.

The lowest level of the reference architecture is the technical platform. This could be a platform like OSGi [80] or SCA [59]. That is a technical platform, which makes implicitly some assumptions about the context of use (e.g., regarding the non-functional requirements). However, this is mostly domain-independent. Such a technical platform can be sub-divided into:

- Service platform infrastructure: This is the basic platform implementation, which cannot be further refined into specific services. This can be realized in an arbitrary (non-service-oriented) way. Thus, arbitrary variability implementation techniques may be applied. Instead of repeating generic surveys on variability implementation, we exclude this from our analysis.
- Technical Platform Services: These are services that are provided from the technical platform. They enable functionality like the registration of services

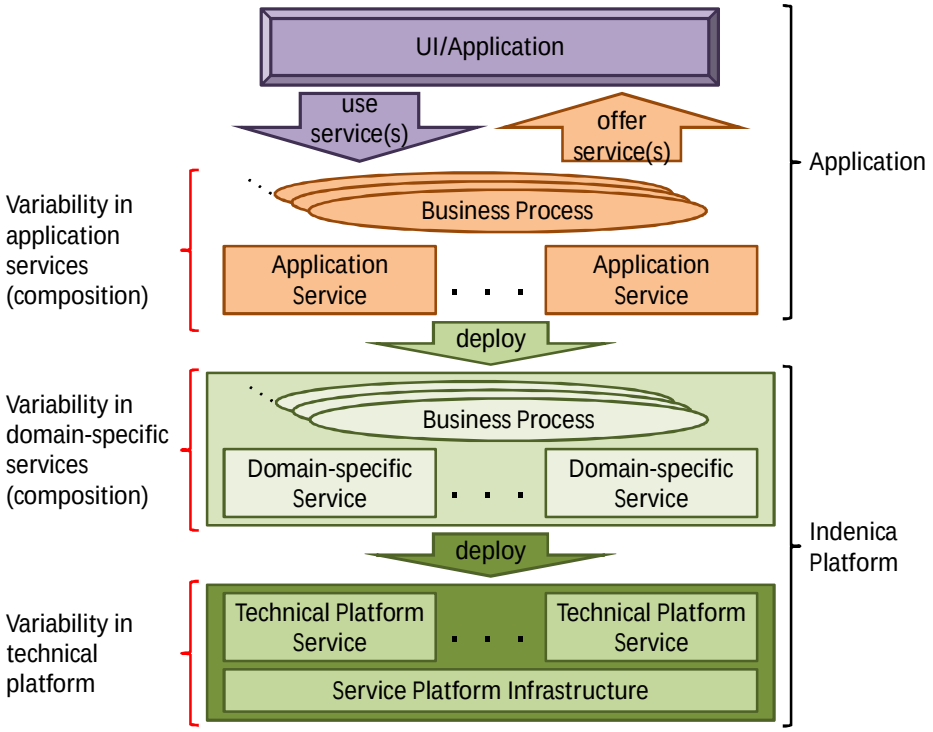

Figure 3: Reference architecture for INDENICA services and service platforms

(provides basis for variability objects)

or other infrastructure capabilities. There can be variability regarding those, e.g., regarding the exact range of services or their exact behaviour.

An INDENICA Platform is a domain-specific platform. This implies in particular that it includes domain-specific capabilities. The range of these capabilities may require customization in turn. These capabilities can be realized as:

- Domain-Specific Services: This includes any variability in domain-specific services where a service is modified, augmented by additional functionality, and otherwise adapted. In particular this may happen either while keeping the interface or modifying the service interface as well.
- Service Composition and Processes: This includes all cases where the specific composition of processes is modified. It encompasses in particular any situations where a specific service is explicitly exchanged for another service satisfying the similar interface, but behaving differently. Service compositions by themselves can form services, but it is worth looking at them independently as usually different approaches are used in such a situation.

Finally, besides varying the composition of functionality it is sometimes also relevant to modify the deployment of services (e.g., changing the number of servers where a deployment happens). Even not deploying a specific service can be used as a way to modifying a service platform. Thus service and platform deployment is the final form of variability object we will look at:

- Service and platform deployment: This covers any form of variability that influences the specific deployment of a service (e.g., not deploying, location of deployment, parameterization, etc.)

Form of Variation

We can differentiate multiple forms of variation. These are rather generic, however, they get a specific meaning, as described below, in the context of service orientation:

- Optional: A variability object may only be part of an installation under certain circumstances. This is called optional variability. This might be a service, or a specific aspect of the functionality of a service or of the underlying platform.
- Alternative: Sometimes it is important that one of several variability objects is present, but the variation is in which of the objects to pick. This can be, for example, one of several possible service realizations that adhere to the same interface or alternative behaviours of a platform infrastructure.
- Multiple selection: Sometimes multiple options from a set of variability objects can be selected.
- Parameterization: Variation can be described also as a parameter, respectively the value.
- Extension: Variation can occur by extending a feature, service or property that already exists. Typically, this is achieved by providing explicit extension points. This can be the possibility to add additional services, or integrate specific functionality in a well-defined form in a call chain.
- Interface: The interface (e.g., of a service) is adapted, e.g. by modifying the number and type of its parameters.

Binding Time

An important question is also: when is the decision made about the specific contents of the variability platform? Many different approaches are possible and discussed in literature. Some examples are given below:

- Implementation time: this means that human intervention is required. In case this variability management is in place, the human intervention might be minimal like changing a few lines of code in a specific file. Thus, it should not be confused with maintenance.
- Compile time: this means that the variability is realized in terms of compile time modifications. Examples could be pre-processors or mechanisms that are part of the build-process like static weaving in aspect-oriented programming. Often a more refined distinction is made, that also identifies link time as a separate binding time or distinguishes between compile-time and precompile-time. This did not proof necessary here.
- Deployment time: the deployment of components or services can be used to realize variability as well. This can happen by identifying whether to deploy at all and where to deploy and also by providing deployment parameters.
- Initialization time: initialization information that is given to a starting system can be used to realize variability as well. In this case, however, the variability must be handled by the system (INDENICA platform) during start-up phase.
- Service-binding time: whenever a service is bound to an implementation, this can be seen as a form of variability, as the same interface may relate to different implementations.

- Runtime: this subsumes all other cases of binding (determining) variability that happens during the execution of the system for which the variability applies.

The list of binding times, which we identified above, only provides a set of reference binding times that we found useful in describing our taxonomy. In any specific system context additional or different binding times can become relevant. There is no single list of binding times, which is useful and relevant throughout all situations.

In the case of runtime bindings, a binding of variation can either be permanent or volatile. A permanent binding is made once during the runtime of the system and henceforth not altered, while a volatile binding is not (in the later case a rebinding for a new variation is possible). Obviously, the distinction between permanent and volatile only makes sense for runtime binding.

In some situations it can be important to support multiple binding times, i.e., it is not only possible to differentiate between different capabilities, but also when the binding of these capabilities is made. However, in such a situation, we need to support multiple variability implementation mechanisms alternatively [72], as any specific mechanism usually only supports one specific binding time.

3.1.2 Context

The context section provides additional information on the constraints under which the variability implementation technique can be applied. The context section consists of the environment context and the assumptions on systems.

Environment Context

This describes any assumptions an approach makes, respectively, technical constraints it has on the environment in which it is used. In particular, a specific approach might be described in the context of Web Services or with OSGi. While this does not mean that it cannot be transferred to situations outside of this context, it only ensures that it is actually working in this context.

Assumptions on Systems

A specific variability implementation technique may also make further assumptions on the specific systems for which it can be used. For example, a technique might focus on being resource-saving and without any real-time disadvantages to make it appropriate for deeply embedded systems.

3.1.3 Solution

The solution section forms together with the purpose section the core part of the pattern template. It consists of six main categories. These are:

Key idea: this describes the key idea of the technique

Technology Background: this describes any general implementation technologies (e.g., aspect-oriented techniques) that are used to implement this approach.

Variability Approach: this characterizes the assumptions the technique makes about the relation of the variable parts and the core parts of the implementation.

Variability Granularity and Selection: this describes the level of granularity on which variability can be described and how individual variable elements can be selected for inclusion or exclusion into a variant realization.

Dependency Management Support: if the technique provides direct support for management of dependencies among variant selection, this is described here.

Platform Definition Support: if the technique supports the identification and definition of the variable parts that are needed to make up a service platform, this is described here.

Key Idea

It is usually difficult to describe all the details of a technology using a predefined schema. This is in particular true as we gathered to a significant extent the relevant information from literature. In such a situation, a description is limited by the amount of information available in literature. This is the reason why we added a free form field to describe the basic idea of the approach to the technology pattern.

Technology Background

While a specific variability implementation technique is typically a rather complex thing and composed of several different elements, there are some fundamental technologies that are used often in the context of variability implementation. Typical examples of this are template processing techniques, textual (or syntax-based) pre-processing with elimination of not required parts, aspect-orientation, etc.

In this section, we link the specific approach to these basic technologies. The expectation is that this may help to identify some general techniques that are used over and over in developing appropriate implementation techniques.

Variability Approach

This describes the basic relation between the core implementation and the variable implementation parts that is used by the technique. The most extreme viewpoints are positive and negative variability, respectively. Approaches that rely on positive variability use an approach that assembles the final realization from multiple pieces. A typical example of this is aspect-oriented programming. Here aspects can be used to implement variable parts. The basic implementation is combined with the aspect implementations to derive the final system. An example of purely negative variability implementation is given by pre-processor-based variability implementation. In such a case a pre-processor is used to cut out those parts of an implementation that are not needed for a specific variant. Other forms of variability realization include generative approaches. In generation-based variability the necessary implementation is generated from a description in a different form. A single variability implementation technique can be composed of several individual implementation approaches.

Variability Granularity and Selection

Different techniques provide different levels of granularity of what can vary and how they identify those parts and their relationship to a specific variation need. Typical examples of granularity are lexical units like a line of code or a file and syntactical units like a statement, parameter, function/method, module, a component, service, service binding, service bundle, deployment unit. Always with the meaning, that this is the smallest granularity supported by the specific technique.

Further in some form the relation between the variable element and the variability decision must be described and also where exactly a specific variability should have an impact. The first is actually often handled externally. For the second different approaches exist. Sometimes annotations are made directly where the variations are (textual pre-processors for conditional compilation are an example) alternatively some generic descriptions can be used to describe where to attach variable parts (e.g., advices in aspect-orientation).

Dependency Management Support

Typically individual variable implementation parts are not independent, but have some relation (e.g., a customization relating to a specific feature can only be done, if another one is done as well). In some cases variability implementation mechanisms have an approach to support this. In most cases, however, an external mechanism is used. This is also one aspect that is typically realized in external tools. In case the technique handles this in an integral way, this is described here.

Platform Definition Support

If many parts are variable in a generic platform, we need an approach identify which parts to select to identify a specific variable platform (i.e., to perform a product definition). Again this is often not done as part of a technique, but relegated to an external mechanism or tool environment. In case, this is part of the mechanism, this is described here.

3.1.4 Further Aspects

In the section Further Aspects those issues relevant to the technique that did not fit in the previous categories are described. In contains only of the entries Source and Comments.

Source

This describes the source of the information on the technique. This might be literature or other forms of information.

Comments

Any information that does not fit the categories above, but is considered important for the purpose of the analysis is described here.

3.2 Variability Implementation Techniques for Service-based Systems

In this section we will provide an overview of the various techniques for variability implementation we found. We categorize current variability implementation techniques in terms of our taxonomy. Therefore, this section is structured according to the variability objects we introduced in Section 3.1.1. Within each section we roughly structure our discussion according to the binding times. A detailed description of the techniques is given as a structured catalogue of techniques in the Appendix of this deliverable in Section A. Within this section, we focus only on providing an overview of the kinds of techniques that have so far been described.

Due to a lack of detailed literature for some of the variability objects, we consider also literature on component-based systems and configurable middleware.

3.2.1 Variability in Service Composition and Processes

In our survey, we identified two representative approaches to realize variability in the composition of services. In Section 3.1.1 we define service composition to be a combination of services.

The composition of services at implementation / compile time is often supported by implementation techniques that exploit MDD techniques, e.g. model transformation and model element mapping.

In [52] the Business Process Family Model (BPFM) is proposed. This represents a variability-enhanced common business process model as a core asset. Park et al. [64] use this as a basis to derive business process variants. The BPFM maps features of an initial feature model to domain activities (these activities are mapped to domain services in a following step). The BPFM is modelled using Unified Modelling Language (UML) [58] activity diagrams that are expanded with variation points, variation point bindings and variant regions. The derivation of a business process variant is based on a given feature configuration and the selection of the corresponding core domain activities and variants. This is, deriving a UML activity diagram in which all variability is resolved. The automated transformation of an UML 2.0 activity diagram to Business Process Execution Language BPEL [63] is presented in [15, 38, 21].Similar to business process models, Steffen et al. [77] introduce reusable flow graphs within METAFrame environment [76] to model service behaviour and then use the final and consistent flow graph to automatically instantiate new services.

Service composition can also be changed at runtime to meet new QoS constraints. Li et al.[46] present a set of algorithms for QoS-driven dynamic reconfiguration of SOA-based systems. In this approach, each Web service is annotated with QoS attributes, e.g. response time or cost. Given a set of available services with their QoS attributes and a new QoS constraint, the presented algorithms will calculate the QoS of the overall system with regard to the process structure. As long as the QoS of the system does not meet the new QoS constraint, one or more services will be replaced to meet the constraint – the overall process structure will not be changed.

SAP provides standard platforms and applications, which can be adapted to the customer's specific business needs. One provided solution is Business Configuration

by scoping and fine-tuning for Business By Design[4], SAPs on-demand platform. Additional to the platform SAP delivers at one hand a comprehensive catalogue (BAC, Business Adaptation Catalogue) exposing the entire set of solution capabilities, described using non-technical business language. At the other hand, SAP and partners deliver so-called Business Configuration (BC) sets, which contains predefined business configurations. At first the customer has to make selection from the BAC based on his specific business needs (scoping), at second he can overwrite parameters of the predefined configuration (fine-tuning). The results are stored in a configuration workspace. The final configuration becomes active through BC deployment. The process comprises activation of UI components and services as well as writing the configuration to Customizing Tables, which are evaluated at runtime.

3.2.2 Variability in Domain-Specific Services

This is the single largest group of patterns we could find. It includes changes to a service that impact its implementation (and thus its behaviour and/or its Quality of Service characteristics), as well as variability of the interface. Though the two forms of variability are closely related, the differentiation between these two types is almost always mentioned in literature due to the higher complexity connected with interface variation. Service interface variability affects always the underlying service implementation in a way so that the provided service functionality has to be modified accordingly. Service implementation variability exclusively affects the underlying methods without modification of the interface.

Variability implementation techniques, which become effective at compile time, exploit the abilities of an underlying, implementation-oriented approach like component composition or application of base asset refinements according to a feature selection. Actually, many of the techniques, we found, focus on implementation time / compile time variability. All of these techniques support the pure implementation variability. The basic idea is in all these cases the same: apply standard variability implementation techniques like component-composition or feature-oriented programming to represent the variability within in an implementation. The component-based service implementation approach [48] focuses on introducing an additional component layer that provides a basis for deriving the service implementation. Like the FOP-based approach, described in [6], it also uses feature modelling as a basis for representing the variability in the implementation. In [76] flow graphs are defined to compose components to a service while a model checker checks consistency among the composition based on a predefined constraint library. The flow graph, whose nodes represent components and whose edges represent conditions under which the related component is activated, is used as input for automatically instantiate the components which yield in the final service. Except for the details of the underlying composition elements the three approaches are rather similar. Somewhat different from these three is the pattern plugin approach [54]. In this approach the focus is on generating the

[4] http://help.sap.com/saphelp_byd30/en/KTP/Software-Components/01200615320100003379/SAP_BBD/SAP_BBD.html

implementation of a configured service from a higher-level description. With respect to the other aspects, the approach is, however, rather similar.

The pattern plugin and FOP-based refinement approaches address also interface variability. However, the approaches focus on direct manipulation of the interfaces (e.g., remove or add a specific parameter). It seems, the approaches do not provide additional guidance to ensure that the interface modifications and the service implementation match. As the modifications are also local to the services, there is no support to ensure that not only the call interface is customized, but also the callee.

Some approaches also support service variability at (initialization time or) runtime. The two approaches we identified differ somewhat in terms of how they handle the creation of a variant at runtime. The class wrapper approach described in [75] is rather similar to FOP-based refinement [6]. The main enhancement that the class wrapper does is that it moves the binding time to runtime. This is achieved by creating a wrapper class, which serves as a proxy and at runtime a new service implementation is created (basically using compile time techniques). These updated implementations are then integrated at runtime using the Java HotSwap [41] implementation. This can be seen as a compiler-oriented approach. Once introduced the new service implementation is directly executed.

The Asset Service Weaver (ASW) pattern [51] has a somewhat different focus. It relies on the aspect service weaver tool [10, 40, 81] to intercept service calls (SOAP messages) and if a message includes a request for a method that the service does not support currently, advice services are queried. These advice services hold the codes for new methods that can be woven into existing services by the ASW. This can be seen as a more interpreter-oriented approach.

SAP provides with the Enhancement and Switch Framework[5] a solution to enhance business functionality of SAPs NetWeaver ABAP core without modifying the original code. The modifications are bound to hooks called Enhancement Options in the original code during runtime, which is similar to Aspect Oriented Programming (AOP). Enhancement Options can be implicit or explicit. Implicit Options are provided by the framework (e.g. begin of methods), explicit options are defined by the developer. There are two kinds of explicit options - source code plug-ins and object plug-ins. With source code plug-ins the developer can define Enhancement Points, which are hooks for adding additional code or Enhancement Sections, which are hooks for overwriting existing code. An object plug-in is called BAdi (Business Add-in). It comprises an interface defining methods, which add new and changeable functionality to an object. At the end the enhancement options can be activated and deactivated with the help of the Switch Framework.

In summary, there exists already a breadth of approaches to implement variability. They cover both development time and runtime adaptation and differ in their approaches and assumptions.

[5] http://help.sap.com/saphelp_nw73/helpdata/en/

3.2.3 Variability in Service and Platform Deployment

The deployment of services, respectively of service platforms impacts the physical layout but not the logical content of a platform. Deployment may influence many characteristics of the running platform at deployment time or even at runtime. In literature, variability of the deployment is usually realized by generative techniques. The artefacts describing the deployment such as models, descriptors or scripts are either generated from additional information in the variability model or by extracting the relevant information from a generic artefact (negative variability).

An example for the first approach is to annotate the variability model with deployment information for each variant. Mietzner et al. describe in [50] that WS-BPEL deployment and undeployment scripts for deployment time as well as for runtime can be generated from the additional information in the variability model.

An example for using generic artefacts is a deployment plan, which describes the deployment of each possible variant including the initial deployment. Ayed and Berbers extend in [9] the standardized CORBA Component Model (CCM) deployment model [55] by variation points in order to express a context-aware deployment plan. At runtime, the generic plan is instantiated by generative techniques to a concrete plan based on context information.

3.2.4 Variability in Technical Platform Services

The services provided by a platform may also be subject to variation. For example this may happen when a platform should be tailored to meet domain-specific requirements, when several platforms are integrated into one and only selected services should be reused or, in INDENICA, when platform services should be composed on a higher level from integrated services.

The techniques for platform service variability in literature typically rely on existing interfaces provided by the (middleware-) platform core. Variabilities are bound to the core platform by implementing selected interfaces and by performing a registration procedure. Table 1 summarizes the implementation techniques described in the remainder of this section. The techniques particularly differ in the individual combination of binding approaches, the use of code generation, the (latest) binding time and whether the technique is integrated with a variability model. Most techniques described in this section bind variabilities at compile time. Four techniques (partially) rely on affecting call chains (interceptors). It is also notable that all described techniques represent the variabilities as some kind of components, in most techniques in the sense of a larger building block with defined interfaces which realizes rich functionality but in one technique also as subordinate parts of a component with simple but distinguishable functionality, so called micro components. This focus is biased by a lack of specific literature for platform variability in service platforms so that we also considered literature on component based systems and configurable middleware.

		Use platform management services	Extension by Interception	Reflective variability	Aspect-oriented composition	Abstract Roles	Application-specific callbacks	Event-based composition	Generated component connectors	Micro-components
Binding approach	Component-based	x	x	x	x	x	x	x	x	x
	AOP				x	x				
	Interception		x	x	x	x				
	Reflection			x						
	Callbacks						x			
Code generation	Variants								x	x
	Glue code							x		
Binding time	compile time				x		x	x	x	x
	runtime	x	x	x		x				
Use of variability model					x			x	x	x

Table 1: Comparison of platform service variability techniques

In the remainder of this section, we discuss first the probably most natural variability technique in service-based systems, the use of platform management services. Then we outline techniques, which rely on interception, then techniques which combine aspect-orientation with interception, then callbacks and finally generative techniques.

Component or service platforms offer management functionality such as suspending, resuming, deploying, adding, binding or deleting components or services. The management functionality can be used as a mechanism to realize runtime variability, either on the level of entire deployment units (e.g. bundles in OSGi) or on entity level within deployment units (e.g. services in OSGi). Parra et al. use the management services of SCA in [65, 66] as variability implementation technique while SAP relies on OSGi.

Additional management services may be added to an existing middleware platform by modifying explicit call chains provided by the platform. In an interceptor chain, a service call is represented as an object and passed to the callee via a chain of interceptors which may modify, consume or reissue the call. Froihofer et al. describe in [31] runtime variability of platform management services by adding, replacing or removing related interceptors.

Reflective capabilities of a component framework offer the potential to reason about the possible variation points and their variants at runtime as well as to dynamically execute the functionality represented by individual variants. Therefore, the reflection mechanism may expose (in [24] on demand) information such as the component interface, the interception chain a component is registered in or the

topology of component instances. In [12], Bencomo et al. apply reflective techniques to realize (runtime) variability for platform services and functionality.

Typically, aspect-oriented techniques (AOT) add variabilities to a small core platform (positive variability). Variabilities are represented as aspects and bound to the core platform using an aspect weaver, i.e. a tool which injects calls to the selected variabilities into the core platform, here at defined extension points. Aspect-oriented composition modifying existing interceptor chains may be applied to integrate and reuse existing middleware services as shown by Walraven et al. in [84]. Extension points of the core platform may also represent additional semantics such as the abstract functional roles introduced by Coleman et al. in [23]. Abstract roles can be used to select dynamically among the services, e.g. based on constraints and QoS-specifications as done in [23].

Alternatively, callbacks can be used to integrate application specific functionality into a given (middleware) platform. As a prerequisite, the platform must provide extension points in terms of callback interfaces, which are then implemented by application-specific functionality in an extra layer on top of the platform. The technique has been applied by Froihofer et al. in [32] to extend platforms such as CORBA, .NET or JBOSS (using remote callbacks over HTTP) with explicit management of application-specific data integrity constraints.

The set of management services provided by a middleware can also be configured using event-based composition and publish-subscribe mechanisms. This technique requires a basic event service provided by the core platform. Middleware services are then realized as components offering their functionality in terms of events. Fuentes and Gamez apply in [33] a middleware variability model for selecting existing services to be integrated with the core and generate the glue code for initializing and registering enabled variabilities.

In addition to glue code also larger parts or even the entire code of a variability may be derived by generative techniques from detailed and precise models. One example is the generation of domain-specific deployment tools by Bures et al. in [18]. A feature-based connector model [17] describes the communication style, the deployment and non-functional properties of the communication of deployed components. The model is used to generate the deployment tool from a code template, the concrete component connectors and the initialization of the connectors in the deployment tool. Another example is to generate only relevant sub-functionality in terms of pluggable sub-modules to be used by a variability. These so called microcomponents are used in [18] to obtain a domain-specific execution environment for component-based systems. Therefore, the sub-activities of the component lifecycle such as starting a component or component lookup areturned into variabilities and represented as microcomponents.

3.3 Summary

As a result of our survey on variability implementation in service-based systems, we identified 20 variability implementation techniques and presented them as high-level descriptions in Section 3.2. In Appendix A, each technique is discussed in more detail

in terms of the taxonomy introduced in Section 3.1. In this section, we summarize our findings and discuss opportunities applying the techniques in INDENICA.

Table 2 summarizes the capabilities of all 20 techniques in terms of the affected variability object, the individual type of variability and the supported binding time. While 70% of the described techniques primarily target one type of variability object, the remaining techniques can be applied to two or even three types. This is particularly true for the technical platform services techniques, as most of them were applied in the context of more general software systems such as middleware or component-based systems rather than being specific to service-based systems. 60% of the implementation techniques rely on previously defined extension points while the remaining techniques focus on optional variabilities and alternatives. In contrast, multiple selection, parameterization and modification of interfaces are additionally supported by only three implementation techniques. Regarding binding time, the described implementation techniques mainly focus on runtime (70%) or compile time (35%). Implementation techniques for deployment also support runtime variability and overlaps between compile time, initialization time and runtime were registered only for technical platform service variability, i.e. few techniques support explicitly multiple binding times. One particular topic for INDENICA, the binding of variabilities based on the QoS is targeted by two approaches, namely Component Service Replacement and Abstract Roles (not depicted in Table 2).

One specific part of the work in WP2 will be to identify relevant variability implementation techniques for INDENICA based on the discussed approaches and to integrate them into a coherent framework. An integrative concept based on production strategies will be outlined in Section 5. Industrial demands for the selection of concrete variability implementation techniques in INDENICA will drive the research (cf. Section 4). Our focus is not to develop yet another set of variability mechanisms for services and service platforms. We rather work on the assumption that all these techniques have their specific role and make sense. What is missing is a more unified approach that will allow integrating them.

As identified above, only few approaches deal with QoS, parameterization or multiple selection, explicitly. Some of the more general techniques for technical platform service variability might also be considered for variability implementation of domain-specific services or service compositions (e.g. callbacks or microcomponents).

		Service Composition Generation	Component Service Replacement	Scoping and fine-tuning	Component-Based Service Impl.	Pattern Plugin	FOP-based Refinement	Class Wrapper	Aspect Service Weaver	Enhancement Options	Deployment / Undeployment Scripts	Context-aware deployment plan	Abstract roles	Application-specific callbacks	Extension by interception	Aspect-oriented composition	Reflective variability	Event-based composition	Generated deployment component connectors	Microcomponents	Use platform management services
Variability object	Platform service												x	x	x	x	x	x	x	x	x
	Domain-specific service			x	x	x	x	x	x				x		x	x	x				x
	Service composition / processes	x	x																		x
	Service and platform deployment										x	x									
Type of variability	Optional	x		x	x	x					x	x						x			
	Alternative	x	x	x	x	x					x	x						x			
	Multiple selection			x																	
	Parameterization			x																	
	Extension						x	x	x				x	x	x	x	x		x	x	x
	Interface					x	x														
Binding time	Implementation time																				
	Compile time	p			p	p	p									p		p	p		
	Deployment time										v	v									
	Initialization time												v		v	p	v			v	
	Runtime		v	v				v	p		v	v	v	p	v	p	v			v	v

Table 2: Summary of variability implementation techniques

(x = supported, p = permanent, v = volatile)

4 Demands for Variability Implementation in INDENICA

In this section, we provide an overview of the main requirements that we could identify for variability implementation in the INDENICA context. There are several sources to these requirements. Some requirements were already described on a high level in the proposal document. Further, we discussed variability implementation requirements with our industrial partners. While these requirements are somewhat influenced by existing products and experiences in building them, they should provide additional insight. The results of this analysis and discussions are described in Section 4.1.

In addition, we analysed some of the platforms that we expect to be used in the use case. The results are also of interest, as we expect that this kind of variation in the service platforms must be supported by our approach. This will be described in Section 4.2.

While we discuss variability requirements here, it should be taken into account that some of the issues are overlapping with variability modelling. In this case, we will only briefly mention this, as we will discuss variability modelling on a detailed level in Deliverable 2.1.

4.1 General Variability Implementation Requirements

Requirements for variability implementation and modelling in the INDENICA-project have many different facets. This includes:

- Typical capabilities that are in general expected from variability realization.
- Issues that relate to the fact that the goal of INDENICA is to customize service platforms and not arbitrary implementations.
- Issues that relate to the fact that it should be possible to derive domain-specific platforms.
- Issues that are derived from non-functional variabilities.
- Issues that relate to supported binding times.

We will now go systematically through these different classes of requirements and will discuss requirements with a specific focus on the industrial partner requirements.

General Capabilities in Variability Implementation

A basic requirement for any variability realization approach is to support the different forms of variation like optional or alternative variation. Most approaches that discuss variability, especially in the context of features, focus only on these variations (e.g. [74]). However, from a practical point of view these are rather restricted. Thus, we emphasize the need to be able to address a larger range of variation. When looking at the various capabilities outlined below, one should take into account that we are not referring to modelling the variability, but modelling the variation. This is very uncommon, as often rather simple approaches that use a direct relation between features and implementation parts are used [43, 68]. Rather, in our

approach, as we will discuss in Section 5, we will allow transformations on the models, based on the variability information that is provided. This is in line with approaches we proposed earlier, like [73]. Next, we will describe the major requirements on handling variable implementation elements:

- *Optional:* an implementation element may only be part of a platform under certain circumstances. This is called *Optional Variability*.
- *Alternative:* Sometimes it is important that one of several implementation elements is present, but it is possible to choose among them. This is referred to as *Alternative Variability*.
- *Multiple selection:* in this case, multiple options exist and arbitrary subsets can be selected. This requires on an implementation level that also the integration / combination of these parts is addressed or is possible.
- *Parameterization:* variation can be communicated through a parameter, respectively a value. On the implementation level it is necessary that this value can be referenced in the correct syntax.
- *Multiplicity:* it might be that some implementation elements need to be repeated. This is the case, if the variability description is providing a variability, but this variability cannot be mapped into a parameter, but rather some implementation element must be repeated. This might be the case if a subpart must be described differently in different elements. Thus, multiplicity may need to be supported in combination with the previous variability operations.
- *Grouping:* the above operations might be applied to whole groups of elements.

In addition to these general cases that are still rather common, it is important that *extensions* are supported, i.e., it is possible to define that something will be augmented by a specific implementation, but it is currently not possible to exactly say what this realization might do. Some industrial partners also explicitly raised this issue. Finally, it is also necessary to address the more general situation of *functionality* that is specific to an individual situation. It must be possible to augment the implementation correspondingly.

A further aspect in variability description is to use more complex languages, like domain-specific expressions and higher-level language constructs. At this point it is not yet clear, whether we will need this within the INDENICA-platform, however, this will be further analysed in the context of D2.1. If this will be included, we will need to introduce corresponding capabilities also on the level of variability implementation. Our corresponding description in Section 5 already takes this into account.

Customizing Service Platforms

The fact that we deal with service platforms had specific significance to some requirements. This leads to the need to address all levels of the service platform model we depicted in Figure 3. Thus, we need to address in particular the implementation of:

- Variability in service composition and processes,
- Variability in domain-specific services,

- Variability in service and platform deployment, and
- Variability in technical platform services.

Within this list the last two are particularly interesting as they strongly embed specific aspects of the service platform world.

Service platforms may vary with respect to the technical platform services that are supported. The way this can be done, however, will depend in general on the specific technical platform and the means it provides. While some platforms provide a basis for this, in other cases we will need to be able to resort to modifying the base implementation.

Also the need to adapt the physical distribution of the platform elements, i.e., their deployment was explicitly raised as a concern. A solution to variability implementation that addresses our needs must also be able to deal with deployment variability. Further, the need to customize service compositions (e.g., business processes) was explicitly mentioned by one of the industrial partners.

In total, it is important to address the whole range of different layers of the service platform taxonomy we identified in Section 3.1.

Domain-Specific Platforms

The need to derive domain-specific platforms indirectly leads to additional requirements. This means, we need to be able to use rather abstract characteristics (e.g., platform for fluid goods) to control the customization. However, this seems less an issue of the variability implementation than of the variability management.

Further, as there is no fixed granularity of what a domain is, respectively, any domain may have sub-domains, it seems reasonable to expect that platforms can also be partially customized (to a broad domain) and successively to a more narrowly defined domain. Such a multi-step approach that allows for partial instantiation will be realized using our approach. This implies that also partial instantiations can be (but need not be) executable platforms.

Currently, it is an open issue, which is under further analysis in the context of the D2.1 deliverable, whether there is a true need to significantly extend the expressiveness of the variability description language and allow more general domain-specific languages in the context of variability specification. It seems that such integration might be beneficial. This would also have ramifications for variability implementation, as the integration of domain-specific languages, which typically rely on code generation, would be required.

Quality Requirements

A major aspect that is traditionally rarely addressed in the area of product line engineering and variability management is the aspect of the variation of quality characteristics. However, in the context of service platforms quality aspects are of very high importance. Based on the information from the industrial partners, the variation of quality aspect also has a very high importance in the context of their platforms. Among others, variation of the following quality characteristics was identified:

- Performance requirements like throughput or latency as well as cycle time
- Real-time requirements; it can vary whether hard, soft or no real-time requirements are relevant
- Scalability requirements may vary strongly among specific cases
- Reliability requirements may vary. Higher reliability demands can, for example, be addressed by increased available redundancy.
- Safety requirements may vary as well, as different subsystems, as well as systems used in different context may vary significantly in terms of their criticality.

Typically quality variation is addressed by varying aspects of the implementation. The major difference that must be addressed is that quality aspects are typically more distributed across an implementation. Thus, our implementation approach needs to be able to cope well with cross-sectional variability implementation.

Binding Times

Various binding times are relevant already today in customizing the various service platforms of the industrial partners. In this context several demands can be derived. Sources of these demands are: the proposal, the context of variability in service platforms, as well as industrial cooperation partners:

- A large range of different binding times exists in service platforms. In particular, the issue of deriving domain-specific platforms requires development time binding of variation, while the nature of service platforms points to the need for runtime binding. Within each of these big areas several binding times exist and in particular deployment binding connects the two.
- Supporting this large range of binding times from an implementation perspective leads to two possibilities: either addressing each (relevant) one separately or providing technologies that enable to abstract from a specific form of binding time implementation.
- It also occurs that the same variability must be decided upon at different binding times. In particular, it happens that the need arises that it must be possible to decide upon a variability either during construction time, at installation time or on a case-by-case basis at runtime.

Thus, we can summarize that a flexible approach that enables a binding time neutral approach to implementation is important.

4.2 Service Platform Capability Variation

In the context of the INDENICA use case, three specific service platforms are expected for integration into a virtual service platform. These are Pococapsule [3], Mobicents [2], and Virgo [4]. On top of these industrial applications will be running. Thus, we did also analyse the variability of these platforms in an effort to ensure that we could also cover corresponding variation on an implementation basis.

In this section, we summarize some of these customization possibilities for these three service platforms, i.e. variability points listed according to their binding times as well as candidates for variability implementation mechanisms that could be used to replicate this behaviour. The variability identified here is only a list of examples

drawn from the current status of the platforms and is supposed to be used as a checklist to ensure that we identified the major possibilities in the previous subsection. Additionally, the variability can be bound at different times in the platform's lifecycle (different binding times for different variabilities), thus we structure the discussion below according to binding times.

Several functionalities may be added or removed statically at least at compile time. Examples for these functionalities are the base configurations of Mobicents and the extension points provided by Pococapsule (plugins for domain specific modelling or supported component frameworks). Here, the extension points provided by the respective platform e.g. in terms of callbacks or even traditional variability implementation techniques such as binary replacement [78] can be applied.

At deployment time, parts of the underlying infrastructure may yet be installed, thus, restricting the number of configuration options, and the deployment of the concrete platform can be selected accordingly. Basically, Mobicents can be installed on different servlet containers whereas Virgo contains a specific servlet container (Apache Tomcat or Jetty) depending on the chosen version of Virgo. However, the underlying servlet container may restrict the available functionality (e.g. in case that MobicentsSip Servlets are installed on Apache Tomcat) or the validity of the installation (Mobicents Media Server requires JBoss application server)[6].

At startup time, functional as well as non-functional runtime properties of the platform can be determined. Some examples are optimization for availability or performance event routing, congestion control or logging amount (in Mobicents) as well as distributed hot-deployment locations, the use of provisioning repositories and further properties such as timeouts (in Virgo).

At runtime, several options of Mobicents and Virgo can be modified using Java Management Extensions (JMX), e.g. controlling OSGi properties in Virgo, and thus, providing means for (adaptive) fine-tuning.

The above cases show examples of variability that are relevant for service platforms in general and for the service platforms that will be used in the project use cases in particular. Thus, the approaches we provide for handling variability should be able to handle the situations above as well – and if possible in a generalized fashion.

4.3 Summary

In this section we discussed what we regard as key requirements for the variability implementation in the INDENICA project. We gathered those requirements both from general discussions of the underlying problem as described already in the proposal as well as using information from the industrial partners on their current (and expected future) situation. Finally, we also validated this with a discussion of some existing service platforms and their customization capabilities. We saw that this existing variability is conformant to the range of variation we described in Section 4.1.

[6] A new version (2.0) provides a standalone version.

We can summarize the major requirements on the variability implementation in the following way:

- It must be possible to cover all typical forms of variability, but on top of this, also case-specific extensions must be easily possible.
- Variability in technical capabilities of service platforms must be well supported.
- Variabilities on all levels of a service platform realization need to be supported (based on the reference model shown in Figure 3).
- Domain-specific variation needs to be well supported, in particular incrementally refined customized would be desirable.
- It must be possible to address quality variation. This requires in particular the need to support cross-cutting concerns.
- It must be possible to support many different binding times, in particular binding times, both from a development time as well as from a runtime perspective. In particular this may apply to the same variability.

All this should be supported in an easy-to-use technology-independent way.

5 Concepts for INDENICA Variability Implementation

In this section, we will describe the major concepts for variability modelling and variability implementation techniques, which will be the basis for the work in INDENICA and in particular for the tool support we will provide. The concepts aim to meet the requirements we identified in the previous section.

A major focus of both, our description here, as well as the variability implementation approach we are developing, is to be technology – or more precisely – artefact-type independent. Towards this end we will introduce the concept of a production strategy, a transformation of an arbitrary type of model (including code as needed) to instantiate variability. This will be the main focus in the first subsection. In the second subsection, we will show how abstraction of binding time and as an effect how a shift of binding time can actually be realized with the concepts we describe here. In subsection 5.3, we will also describe how to formulate the special case of a service-technology independent variability implementation using the concepts described in this section.

5.1 Production Strategies

In the previous section, we introduced the major requirements on variability implementation that can be derived from the context of INDENICA. A major requirement was to be rather independent of the specific service technology. This is achieved mainly through introducing the concept of a production strategy.

A production strategy defines how variant parts must be assembled in the presence of a variability resolution (i.e., a value was assigned to a decision variable). To be more general, arbitrary expressions over decision variables may provide the basis for driving the instantiation of the variability. Thus, a production strategy takes several partial source models (or in general parts of artefacts) of some kind and realizes a selection at a specific variation point in a target model as illustrated in Figure 4. As shown there the introduction of the selected element(s) can be combined with the generation of additional glue code. A simple example of a production strategy is realized by the `#ifdef` in conditional compilation in the C language along with the preprocessor that interprets it. This can be used to realize a variability in the context of the C programming language. Although, preprocessor macros have negative ram-

Figure 4: The basic concept of a production strategy

ifications [53], they are often used in practice and can serve here to illustrate our notion of a production strategy. We use this mainly because it is simple and widely known, not because we particularly recommend this technology. In this case the elements are the parts guarded by the `#ifdef`, `#else`, and `#endif` constructs, the selector (the specification how sub-elements are defined and combined) is exactly formed by these constructs, while the variation point is provided by the position where this construct is placed. In this case no glue-code is produced because the glue is already defined in terms of source code, i.e. only the selected element becomes part of the target model, which here of the same type than the source model (C language source code). The selector also provides the necessary capabilities to determine which of the elements to put. While this example is very simple, most real cases will be more complex, but can also be described using this approach.

Any production strategy consists of certain elements in order to realize a variability. These elements can be easily identified, if we analyze what a production strategy must achieve. A production strategy attaches to a certain point in a model (also source code is seen as a model here), and instantiates the model by selecting some elements that are related to this variation point (based on information about chosen values for the decisions), combining them in some form (if necessary) and inserts the result at the variation point. The combination of the elements is necessary, e.g., if we select multiple values. In this case some glue needs to be produced in order to identify at runtime the necessary element. However, the notion of glue is meant here more generic. It can basically be any kind of additional model elements, required in order to embed the selection result. The details of what is selected depend on the chosen decision values. Thus, we can say, in order to describe a production strategy, we need the following elements:

- *Definition and evaluation of a variability value:* a way of mapping decision values that are used to define a specific product into specific resolutions that include the variant parts and include them appropriately into the target artefact. Using the example of the C-Preprocessor, `#ifdef` directly takes an argument which must be provided by the environment.
- *Variation point identification:* this describes where the variation occurs, in particular, where the result of the production and selection process is placed. In the example, this is given by the textual context of an `#ifdef` statement in the source code. However, many different means exist for identifying this. This will always depend on the available technologies and the relevant types of artefacts.
- *Technique for selecting (and combining) elements*: this defines how sub-elements (e.g., the various *alternative* elements of a selector) are identified and combined. A combination is necessary in case of a multiple selection. In this case, the various selected sub-elements must be integrated in some way. This also subsumed as glue in Figure 4.
- *Technique for introducing selected elements (including relevant glue)*: after selecting and combining the final elements, it might be necessary to use some sort of insertion mechanism in order to integrate the combined parts

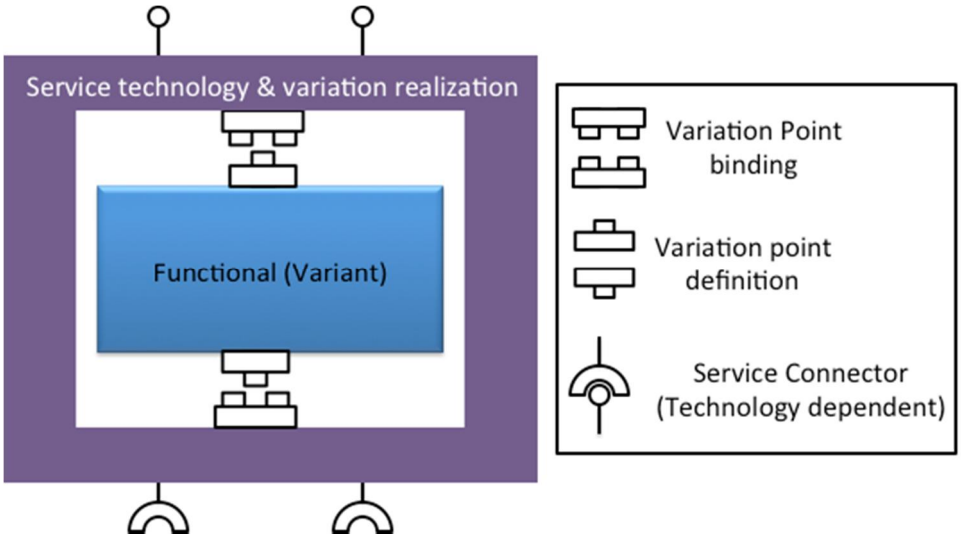

Figure 5: Separation of functional code and variability implementation

into the target model (at the position denoted by the variability point identification). An approach, which requires this, is when aspect-orientation is used and the individual functionality that varies is described using aspects. In this case, the aspect preprocessor provides also the capabilities for inserting the result.

As outlined above, a single realization technique (like a preprocessor) is often insufficient to address all variability concerns. This is the case even if we restrict ourselves to a specific form of representation (like C-code), a specific form of selection (e.g., multiple selection), and a specific binding time (e.g., preprocessing). It is, for example, problematic to use C-preprocessor directives to realize multiple selection, as this requires glue among the selected variabilities.

In order to successfully support variability implementation in the way we discussed it in Section 3, we need also to realize the artefacts in a way that is independent of the specifics of what we would like to insert. So, in order to provide technology independent variability implementations, i.e. the ability of exchanging the service technology while applying the production strategy, we also need to abstract in the underlying artefacts from this.[7] This form of separation is shown in Figure 5.

In Figure 6, we show how the implementation of a production strategy may look like in a process view. Based on the decision model a (partial) instantiation is described. These values must be translated into a form that can be understood by the selector. This selector definition is then the basis (together with the variation point definition) to derive with the help of the glue generator an implementation. The resulting implementation is then injected at the variation point into the target model.

In order to more formally – and thus, precisely – define what a production strategy is, we need to look at its characteristics. These include: the type of target model (e.g., C-code), the binding time (e.g., preprocessing), and, the kind of selection supported. Nevertheless, a production strategy is still generic. It still provides a very

[7]We combine here the realization of variability with the service technology realization. It is currently an open issue whether it might be better to vary the service technology in an independent step.

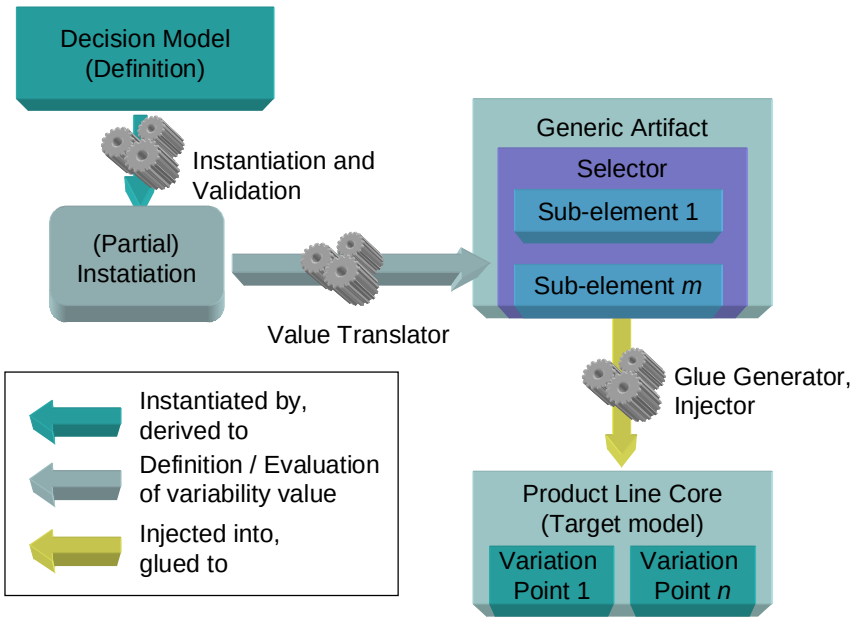

Figure 6: Implementation of production strategies

general mechanism that can be applied to a very large number of variabilities in a product line. In summary, we can interpret an individual production strategy as a transformation function of the form:

$$PS(target, vip, val, elem_1, \ldots, elem_n) \rightarrow model$$

Here, *target* provides the target model where a variability must be resolved. The specific point in the model where the results of the variability resolution shall be inserted is also called a *variability impact point* (*vip*) [70]. *Val* defines the value of the decision for a specific product. Finally, $elem_1$ to $elem_n$ are the elements among which a selection occurs, i.e., the arguments to the variability selector. This can be used to more formally describe the generation of instantiated platforms. However, a complete formalization still requires much further work.

While Figure 6 provides the view of a single application of a production strategy, Figure 7 provides a more comprehensive view of the production process as a whole as it is currently seen. The basic implementation that provides the functional content (depicted as base production code) is transformed using production strategies. In this case, we expect that variability code which is binding time specific will actually be introduced through this operation. This leads to integrated production code, which contains the actual variability (and service technology) implementations. This code is then exposed to the different production stages, leading to the binding of variabilities where appropriate.[8]

5.2 Binding Time Shift as Exchanging Production Strategies

The selection of the implementation technique for a variability typically fixes the binding time. However, sometimes, this is undesirable and it is desired that the

[8]Note that this is not necessarily done this way, actually the introduction of the production strategies could happen at exactly those stages corresponding to the binding times. However, this would be more complex. The depicted approach has also been applied in previous work [72].

binding of platform services can also be flexible in their binding time. For example, for one instantiation of a platform it is known that several services should not be present at all, while for another instantiation of the same platform the decision on the available services should be made dynamically at runtime.

Examples of approaches that support such variability of the binding time are timeline variability [28, 29], anytime variability [82], or meta-variability [71, 72]. These approaches support a dynamic shift of the binding time. Timeline variability and anytime variability are very restrictive, as they consider only a specific form of variability realization.

Experiments showing the technical feasibility of the generation of glue code and shifting binding times have been implemented in a prototypical tool [71, 72]. In fact, the tool described below is early work in this field based on initial concepts that predate the project. The concepts we introduced in the previous subsection provide a basis for a further systematization of this approach. As part of the further work in the INDENICA-project WP 2, this will be subject to thorough analysis.

In this initial tool-based approach the variability model as well as all artefacts relevant to the product line, like source code files, serve as an input. The output is the product, which is built according to the variability selection as an instance of the variability model. In this case the variability information also includes the binding time information. Based on this information, the tool configures the build process so that only relevant parts of the artefacts are composed, compiled and packaged.

Based on the terminology introduced above, one could reinterpret this early tool as realizing multiple production strategies, where each production strategy is realized as an aspect template. Based on annotations in the target code (we use Java as a programming language in our examples), the tool determines where and which variabilities need to be actually realized for a specific system. Using this information, aspect templates are instantiated and woven into the final production code. In this way, we only have a single template per production strategy. Moreover, all

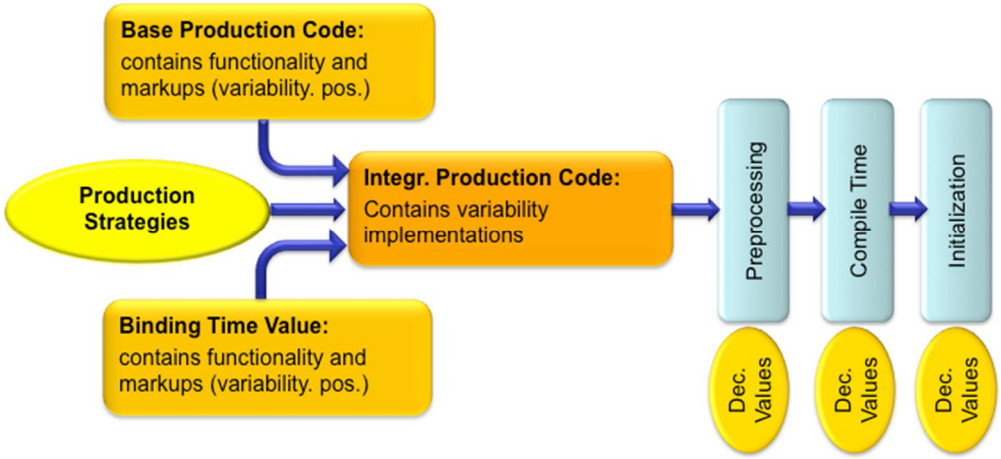

Figure 7: The production process

production strategies are handled in the same way and are completely independent of the functional code. This approach can be seen as an early attempt into the direction of binding time variability, as it is expected to be supported by the

```
1  aspect <<GlueClass>> {
2
3      pointcut targetClass(): within(MainWindow);
4
5      // replace factory call
6      pointcut componentFactory(): targetClass()
7      && call(void <<ComponentName>>.create(JFrame, JTabbedPane,Client))
8      && if (!variability.StaticConfiguration.<<DecisionId>>);
9
10     // just delete the component constructor
11     void around(): componentFactory() { }
12
13     // avoid multiple weaving
14     static {
15         ProcessingState.setKey("addMethod", "<<GlueClass>>");
16     }
17 }
```

Figure 8: Aspect template for compile time binding

INDENICA project. However, this initial approach was restricted, for example, in the sense that only aspect-based injection of variability implementation techniques was possible and thus also only the handling of java artefacts was possible. Further, this initial approach was not aligned specifically with the context of service technologies. Moreover, the specific systems on which it was applied had to be particularly prepared, but no architectural modularization like the one we described in the previous section was used. However, this initial study showed that the implementation of a binding time shift for compile time, system initialization, and runtime is feasible without too high demands on the specific, underlying software.

An example for a compile time binding template is shown in Figure 8. In this example the template parameters are determined with the help of source code annotations. Using these parameters the template code is instantiated and an aspect results that takes care of the specific variability for which it was generated. In order to address the variability (in this case an optionality) we remove the creation of the instance and its registration from the system (line 11). As can be seen in the figure (line 3 and 6-8), the template makes certain assumptions about how the various variabilities are interwoven with the overall code. To some extend these dependencies are due to the specific realization approach we use for the production strategies (i.e., aspect orientation, using the AspectJ processor), and to some extend this is due to fundamental issues, as we need some conventions for variations, as otherwise the insertion or deletion of code might lead to semantically or functional incorrect code. In the future we expect to address these assumptions in a more systematic manner than is the case in this early prototype.

Similar to the aspect template, we showed in Figure 8, we realized other aspect templates for initialization time binding and runtime binding. In these cases also additional code for handling dependency and configuration issues is currently generated in an ad-hoc fashion. Again such an insertion is not possible without making some assumptions with respect to the underlying system. However, we achieved complete separation of the variability mechanisms from the functional code in the sense that we can arbitrarily exchange the various variability implementations for different binding times. Thus, we regard these experiments as a

promising basis for extending our work along the lines of the described general production strategies.

The early approach described above has several restrictions:

1) It relies on an aspect-oriented implication (currently AspectJ). This technology is so far only little used in practice. Moreover it makes strong assumptions regarding the underlying implementation and the kind of modifications. Within WP2 we expect to be able to modify the approach so a larger range of systems and situations can be addressed.
2) The preparation of the system was done manually by inserting source code annotations or by changing the build process. This required deep knowledge of the code as well as of the dependencies to the tool. Here, we expect for the future an approach that is easier to handle. However, an explicit definition of variability points will always be necessary as an understanding of the semantics of the variation is required.
3) As stated above, the approach can be interpreted as applying multiple production strategies. In fact, the concept of a production strategy is not realized in the early tool prototype and, thus, it is difficult to generalize the application to other kinds of systems. For the full version to be developed in INDENICA we have to face these problems, i.e. to provide a clear separation of technical glue code, to consider consistency issues (e.g. passivation or state transfer) and functionality as well as to integrate the approach with architectural concepts.

A summary of the production strategies for the three different binding times outlined above is given in Table 3.

Production strategy element	AspectJ production strategies for shifting binding times
Definition and evaluation of a variability value	Source code annotations mark the variable parts in source code and link to the variability model. Generated attributes represent the individual variability bindings. Bindings are represented at • Compile time as constants • Startup time or runtime as variables In the aspect templates, pointcut specifications or aspect source code representing the glue code may refer to these attributes.
Variation point identification	AspectJ pointcuts guided by source code annotations.
Technique for selecting (and combining) elements	The variability model defines valid binding times per variability (meta-variability). The concrete selection defines the binding time to be applied as a one-out-of-many selection and thus the concrete production strategy. Multiple selection is not possible.

Technique for introducing selected elements (including relevant glue)	A code generator produces the attributes containing the variability bindings. Depending on the binding time, the template processor selects the concrete template and instantiates it using information from the variability model and the code annotations. Product instantiation happens by weaving the generated aspects into the system. • Compile time: static weaving • Startup time: dynamic weaving, variability binding at system startup showing a UI dialog • Runtime: dynamic weaving, add a menu item for the variability binding dialog, display the UI dialog on request

Table 3: Summary of the production strategies for compile time, startup time and runtime.

5.3 Variability of Service and Component Technologies as an Exchange of Production Strategies

In INDENICA, the integration of multiple platforms that use a number of different (service) technologies is a major goal. This implies that specific functionality should be flexible to be integrated with different component and service implementation technologies. However, according to the state of practice, varying an implementation with respect to its component or service technology is extremely difficult. In an ideal environment, at development time, the functionality of a service should be in the focus of the software development while the technical integration with the execution environment (middleware, service platform) should be transparent to the developer, e.g. by automation. However, today each technology provides different capabilities, e.g. service call styles and requires specific additional information or even a certain implementation style, e.g. annotations for Web Services, IDL in CORBA or specific type compliance in RMI. This variation in implementation approaches makes a combination of technologies or a migration among technologies rather difficult. This, in turn, prevents the reuse of the functional realization across service technologies. In order to improve reuse, we aim at abstracting, as far as possible from these technology specific issues.

In the following subsections we first introduce our view on service and component technologies. We will focus on the perspective of varying the implementation technology as a form of meta-variability. In Section 5.3.2, we discuss the commonalities and differences of service technologies as this determines the limits on the extent to which it is possible to exchange service technologies. Finally, we describe an example of an exchange of service technologies in terms of production strategies.

5.3.1 Service Technology Fundamentals

A *service technology* provides a specific way of defining and using services. Service technologies may target specific application domains, such as web information systems (e.g. Web services), facilitate dynamic loading and reuse (e.g. OSGi), provide storage and lifecycle management (e.g. SCA) or remote invocation (e.g. SCA, OSGi). While *service-based approaches* lead to a different structuring of the system,

compared to classical *component-based approaches*, the underlying capabilities are to a large extend comparable (as we will discuss in Section 5.3.2). One could thus also employ an approach like CORBA to realize a service-oriented implementation, although the typical technology for doing so is probably Web Services. For this reason, we will use the term *service technology* to mean any form of technology that supports the interaction of parts of (distributed) software systems.[9]

In order to facilitate the seamless exchange of service technologies, a set of properties specific to implementation technologies must be encapsulated and hidden from the calling program. We call this set of properties a *technology profile.* It describes an abstraction from the underlying capabilities of a service technology and thus provides a means for a comparison of technologies. The technology profile in particular includes the set of technical platform services as defined in Section 3.1.1. A service technology may:

- Require a service interface or implementation to comply with a certain type, e.g. the bundle activator interface in OSGi. Such technology-specific parts of the implementation can be encapsulated in service wrappers [35].

- Restrict the use of parameter and return types, e.g. String arrays need to be represented as specific class instances in certain Web Service implementations. Thus, it might be necessary to specify explicit value conversions for a certain implementation technology. More sophisticated mappings are needed, if method names or parameter sequences cannot be directly mapped.

- Provide a specialized lifecycle management for the components and services. Often the instantiation depends on specific mechanisms of the implementation technology, e.g. in SCA instance creation can be used while in Web service technologies an appropriate stub of the service must be created.

- Ship with different sets of management services so that the technology profile may specify the management services being relevant for exchanging implementation technologies.

5.3.2 Comparison of Service Technologies

We analyzed several service and component technologies as a basis for determining the limits to which portability across service technologies would be possible and hence limits to abstract from specific service technologies using production strategies.
In literature, several surveys classify and compare up to 30 different service-oriented and component-based technologies [16, 25], respectively up to 23 component-based technologies [26, 45]. We use the criteria applied in literature as a basis for our analysis and refine them by additional (technical) categories according to the focus

[9] Typically, service-orientation is connected with a much higher degree of run-time flexibility as component-based technologies. However, in practice runtime discovery is also possible with classical „component-technologies“ as well as service technologies are often used in a more static way.

of INDENICA and our specific task of understanding the possibility of varying service technologies. In the remainder of this section, we present a subset of the categories and technologies considered in our analysis, particularly those related to the technology profiles introduced in Section 5.3.1 and to management services relevant to variability implementation techniques, namely:

- Capabilities related to technology profiles
  - The type of (service) units supported by the technology as well as possible dependencies among them for defining compositions [26, 45].
  - Capabilities for describing interfaces to the supported units [16, 25, 45].
  - Call semantics and interaction style provided by the respective technology [26, 45]. This is particularly relevant for separating service functionality from technology integration.
- Supported platform management services:
  - Query and discovery of deployed units; these are considered typical management functionality in service-based systems.
  - Introspection of units [16, 25], i.e. dynamically obtaining the capabilities of a unit and providing the basis for reflective variability implementation techniques.
  - Specification and support for extra-functional properties [26] such as quality of service or security.
  - Functionality for deploying units [26].
  - Event mechanisms for communicating among the units or to receive information on the state of the platform.

Regarding the considered service and component technologies, we rely here on the capabilities as they are described by official specification documents and do not take specific functionality of individual implementations into account. We included those technologies, which we considered particularly relevant to INDENICA such as the Service Component Architecture (SCA) [59] (which is under discussion as a foundation for the INDENICA runtime environment) and OSGi [80] (realized by Virgo [4]). Further, we include other well-known technologies, which are also considered in the surveys mentioned above, such as web services the Common Object Request Broker Architecture (CORBA) [57] and Java Remote Method Invocation (RMI) [60]. As one kind of web service technology we refer to the web service stack as specified by OASIS, i.e. the Simple Object Access Protocol (SOAP) [62, 86], the Web Service Description Language (WSDL) [85] and the Universal Description Discovery and Integration (UDDI) [30].

Capabilities			SCA	OSGi	Web-services	CORBA	RMI
concepts	basic unit type		component, service	bundle, service	XML-Messages	class, struct, module	Remote Interface, Remote Object
interface	interface relations		provided, required	provided, required	provided	provided	provided
	dependencies		wiring	wiring	via WSDL	remote reference	remote reference
	explicit rules	rule types	intents, policies	bundle specification, intents	-	policies in IDL (extension)	-
		Binding time	assembly, deployment	(runtime)	-	assembly, (runtime)	-
	explicit protocols		conversions, transactions	-	-	transactions	-
Call semantics	remoting	parameter type	by value, by reference (in same process)	by value	by value	by value, by reference	by value, by reference
		callbacks	possible	possible	-	possible	possible
	local calls		possible	possible	-	possible	-

Table 4: Service technology capabilities related to technology profiles.

Error! Reference source not found. summarizes the categories related to technology profiles, i.e. basic unit types, interface capabilities and the call semantics. The categories discussed above are refined into sub-categories in order to highlight selected capabilities found while analyzing the individual technologies. Details on management services are summarized in a second table below.

The *basic unit type* denotes which kinds of units are supported by the platform. SCA[10], OSGi, and CORBA use mainly services, bundles, and modules. Web services as well as Java RMI are intended for Remote Procedure Calls (RPC) and either rely on messages or special types of objects. More specifically, for Web services SOAP differentiates between such RPCs and sending complete XML documents.

Interfaces to the units handled by a platform are the primary artefact to be considered for the variability of the service technology. While SCA and OSGi support the explicit specification of *provided and required* interfaces, the remaining technologies typically support only provided interfaces and handle required interfaces implicitly (e.g. via class loading in RMI). SCA, OSGi and Web services (using WSDL) provide explicit mechanisms for modelling *dependencies* among units outside their implementation while CORBA and RMI rely on remote references used in the

[10]See also the discussion on nested elements in INDENICA Deliverable D3.1.

implementation. The interfaces as well as the wiring (if supported by the technology) may be described or constrained further by *rules*. For this, SCA offers two types, namely policies (capability description or constraints) and intents (refinement of policies in terms of requirements). Initially, OSGi supports only rules regarding its deployment units (e.g. bundle version, bundle dependencies) while recent versions of the OSGi specification also define intents similar to SCA. Also the basic CORBA versions did not support rules while recent extensions of the standard add policies (here a characteristic possibly shared by concrete objects) to the Interface Definition Language (IDL), e.g. in [56]. Rules may be specified at a certain *binding time*, i.e. the point in the application lifecycle, where the units and their connections must at least be established. Furthermore, two technologies allow the specification of protocol styles: SCA supports explicit conversations (fixed communication sequences) as well as transactions following the ACID (Atomicity, Consistency, Isolation, Durability) paradigm while CORBA supports only transactions.

The *call semantics* section indicates the supported interaction style, i.e. whether remote calls or explicit local calls are possible. All technologies support remote calls as well as passing parameters by value. SCA, CORBA and RMI also support passing parameters as references; in the later case additional restrictions apply for SCA. All technologies except Web services[11] support callbacks, i.e. the registration of a local unit at remote side to be called from the remote at defined extension points. SCA, OSGi and CORBA support explicit remote calls, i.e. the capability to replace remote calls by direct calls to locally hosted units in order to avoid network overhead and to improve performance.

[11] C.f. SectionA.4.2 for work on callbacks via HTTP as a possible basis for callbacks in Web services.

Capabilities		SCA	OSGi	Web Services	CORBA	RMI
query & discover		by domain-level composite, name or URI	by name, URI	UDDI, Web Services Dynamic Discovery	by name, repository interface	by name
introspection		-	-	-	via get_interface, repository interface, CORBA reflection, ORB interface	-
security		via policies	permissions	additional specifications	security policies, trusted domains	---
QoS		via intent	bundle capabilities	-	policies, fault tolerance	-
deployment functions		install, update, remove, add / remove composite	install, start, stop, deactivate, update, uninstall	-	bind, rebind, unbind, upgrade	export, bind, rebind, lookup, unexport
events	unit	yes, in channels	yes	-	yes, as defined in IDL	-
	platform	-	yes	-	-	-
	event service	yes			yes	

Table 5: Management services provided by service and component technologies.

Error! Reference source not found. summarizes management capabilities of the considered technologies. Each technology provides individual capabilities for *querying and discovering* units, e.g. using a qualified name. In addition, the CORBA specification describes an interface to the internal repository, which may be used to execute queries. In CORBA, the same repository interface as well as a specific reflection interface or the Object Request Broker (ORB) interface may be used for *introspecting* units. The remaining technologies do not explicitly support introspection (this might be handled using the reflective capabilities of the underlying programming language if available).

Except for RMI, where the basic Java security mechanisms apply, all techniques provide explicit security mechanisms, in additional specifications (Web services), on source code or bundle level (OSGi) or on interface specification level (SCA, CORBA).

Only SCA, OSGi and CORBA provide (basic) mechanisms for specifying and enforcing Quality of Service aspects. While the Web service specification does not explicitly define management services for the deployment (this depends on the implementing container), the remaining technologies define similar capabilities for installing, starting, stopping, updating or deleting the respective units.

Some technologies offer information on the execution of management actions or on the change of certain unit states. While SCA, OSGi and CORBA provide events on their respective units, only OSGi also offers events describing the platform state. SCA and CORBA provide a general event service, which might be used to realize event-based variability (c.f. Section A.4.6).

In summary, we compared the individual capabilities of five different technologies: SCA, OSGi and Web services as service-based platforms as well as CORBA and RMI as (more) component-based platforms. While this analysis does not go into the technical details, e.g. differences among policies in SCA, OSGi and CORBA, it provides an overview on the main capabilities and the differences among these technologies. As these differences provide major obstacles in exchanging one service technology for another (if a relevant management service is not available, it basically makes the exchange impossible), this also describes the potential for varying service and component technologies.
The specifications of SCA and OSGi define various similar concepts, particularly since the recent alignment of OSGi to SCA. Based on a direct comparison, OSGi appears to be the more basic (but evolving) service platform while SCA is more advanced. However, CORBA shares a lot of the discussed concepts with both SCA and OSGi, but also differences exist like in interface specifications or with respect to introspection.

5.3.3 Using Production Strategies to Abstract from Service Technologies: an Example

As mentioned above, the generation of wrappers for a given interface (such as stubs and skeletons) is state of the practice at least since the introduction of CORBA and is used to translate WSDL to concrete programming language constructs. Also other approaches target the generation of wrappers among components, e.g. Cavallaro et al. [22] automatically produce wrappers between two different WSDL web services or Zhao et al. [87] generate glue code and component wrappers to achieve interoperable computing systems by unifying the communication to SOAP. Our work goes beyond existing approaches because we abstract from the concrete technology and generate the code for the integration technology in a rather general form. This allows the exchange of the service technology on demand using product line techniques.

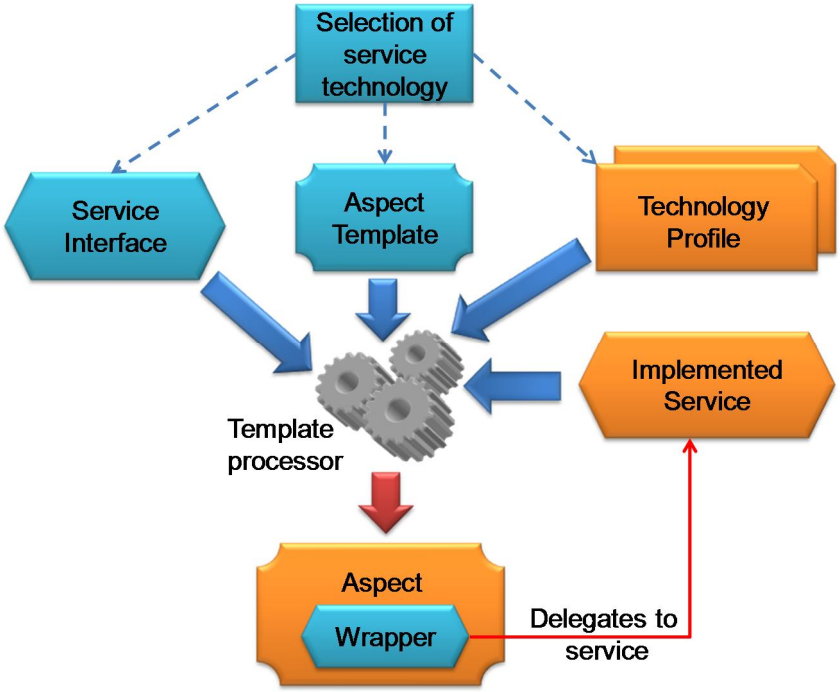

Figure 9: Overview on generating aspects and wrappers.

In a prototypical realization that aims at demonstrating the feasibility of independence of the service technology, we varied the specific service technology using aspect-oriented techniques to inject the implementation of an individual instance of a service technology into the target system. In the prototype, we use aspect-orientation merely as a kind of preprocessor. To render our approach independent of the specific system under study, the concrete aspects are generated from a set of basic aspect templates. Figure 9 depicts an overview of the generation process in our prototype.

Once a specific instance is desired, the affected service interface, the related aspect template containing the generic realization of the binding, and the service technology (profile) are selected. The code generator takes this information as input and generates the glue code for binding the variability, similar to the approach for shifting binding times in Section 5.2, as well as a wrapper according to the appropriate technology profile. The aspect is then used to inject the realization of the service technology for the specific decision into the target system.

While this approach enables independence from the specific service technology and thus also supports the exchange of service technologies, it does not modify the underlying architecture. Thus, if a true component-based or service-oriented implementation is desired, corresponding services must be designed and identified on an abstract level. The specific technology, however, can be determined and changed at a rather late point in time.

A summary of the elements of a production strategy that realize a service technology is given inError! Reference source not found.. Individual strategies may differ in the selected technology profile and depend on the concrete service interface they are applied to.

Production strategy element	Example production strategy
Definition and evaluation of variability value	Source code annotations mark the variable. Source code annotations mark the variabilities, each with the identification of the realized variability. The tool environment matches the identification against the variability model.
Variation point identification	Variation points are implicitly represented by the interface of the services marked as variabilities.
Technique for selecting (and combining) elements	The variability model defines the available service technologies as an alternative. The concrete selection defines the technology (profile) to be applied as a one-out-of-many selection. Multiple selection is not possible.
Technique for introducing selected elements (including relevant glue)	A code generator produces the technical integration into a service technology based on the concrete technology profile, glue code templates and the individual service interface. Particularly, in case of remote interfaces, the generator produces the appropriate call style, i.e. by reference (e.g. as remote callback similar to the variability implementation technique in Section A.4.2) or by value.

Table 6: Summary of production strategy elements for service technology independence

5.4 Summary

In this section, we introduced the key concepts relevant to variability implementation according to the INDENICA approach. The core aspect is the introduction of what we call a production strategy. This provides a generalized means of introducing a variability implementation into an asset.

In particular, we propose to separate on the one hand the specifics of the service technology from the functional aspects and on the other hand the variability implementation from the functional aspects. This enables to introduce variability implementation and service technology on demand and thus provides a significant degree of freedom, which is otherwise not possible.

The effects of such an approach, we studied using two different examples. First of all, by separating the variability technique, we can introduce at a very late point in time the necessary variability. This allows, among other things, to vary the effective binding time very late. However, it can also be used to separate out the service technology and thus introduce a specific service technology at a very late point in time.

We also discussed limits to such an exchange of variability techniques, by analysing the differences among individual service technologies. This provided some insight into which technologies will actually be exchangeable.

Within the further work in INDENICA, a particular focus will be on:

- demonstrating the described forms of variability realization in further, more complex examples.
- formalizing the notion of production strategy, in order to analyse in more depth its applicability and usefulness.
- integrating and extending this approach in an integrated tool environment.

6 Conclusion

In this deliverable, we aimed at deriving the initial concepts for the variability implementation approach that will be used within INDENICA to support domain-specific customization of service platforms.

In a first step, we summarized main concepts that are relevant to variability implementation in general. As of now, we assume that we will be able to address most concerns using a configuration-based approach. Thus, the focus was on a decision-oriented description of variability, which provides the basis for the effective tailoring of the service platforms. We will study this further in the upcoming D2.1 deliverable.

In Section 3, we studied in detail related work. The focus was less on a positioning of this work relative to other research as it was clear from the beginning that the broad focus that we have in this work is so far unmatched. Rather, the main focus was on providing a collection of best practices from research and industry that we would like to include our at least the capabilities of which we would like to emulate as part of this deliverable. While Section 3 summarizes these patterns, the full description of these patterns is provided in Appendix A.

Section 4 focused on capturing the main requirements that any approach to variability implementation that addresses the INDENICA needs, must provide. As part of this work, we could identify a number of different requirements. Some had already been described (on an abstract level) in the project description: However, a number of requirements could only be identified through interviews of the industrial partners and through analysis of the state of technology, including the expected platforms that will be used as part of the use case. A major challenge will be to strongly abstract from the specific service platform technology, address the broad range of different binding times and still be able to provide a customization approach that can easily be used.

Section 5 then focussed on the concepts we developed for achieving a new approach for variability implementation. The main contribution is to provide a significant level of abstraction that enables to go beyond the flexibility most current approaches to variability implementation provide. This relies on earlier work which shows that binding time variability can be achieved using this approach. But we extended it here both conceptually as well as in demonstrating the applicability to different service technologies.

However, while this deliverable established some core concepts of the INDENICA variability implementation approach, still a lot of work needs to be done. We need to further formalize the implementation approach in order to provide a more generalizable applicability. It is not yet fully clear, but this might require also a specific form of modelling the services together with the variability. Also the resulting production strategies need to be implemented in the tool environment, in order to demonstrate and test the applicability of the approach. At this point the tool only provides rather limited production approaches.

Also the variability in terms of quality of service aspects will provide a challenge, where we expect that further work up to the final implementation deliverable needs to be done both on a conceptual as well as on an implementation level.

Finally, a question, which we will address in the context of deliverable D2.1 will have major ramifications also for the further work in variability implementation. In D2.1 we will discuss in detail the potential approaches for modelling variability per se within INDENICA along with its trade-offs. The specific form(s) of variability modelling that will be supported, may also have a significant impact on variability implementation.

A Appendix: Variability Implementation Patterns

In this section we provide an overview of different variability implementation techniques found in literature, which are relevant to service oriented development. Our overview is mainly based on a literature review, but also takes into account reported practices from project partners.

The structure of our patterns is inspired by standard pattern catalogues like [35], however, it was refined based on the specific context of our work, namely variability implementation and service orientation. A more detailed description of the structure is provided in Section 3.1.

Name: Each form of handling variability implementation (aka variability implementation pattern) receives a unique name for easy identification and reference.

Purpose: The variability implementation problem that the technique supports to solve is described in this section of the template.

Description: A short description on what problem in variability of service-based systems and service platforms the variability implementation technique aims to solve.

Variability Object: This describes the object that is supposed to vary. In Accordance with Figure 3 and the discussion in Section 3.1, we differentiate between:

- *Technical infrastructure:* the technical infrastructure provides the backbone of the service platform. As this is typically not by itself service-based, any form of variability implementation can be used with this. In order to restrict the scope of this survey, we do not address this here.
- *Platform service:* within a service platform also infrastructure services exist like the registration of services, location of services, etc. A platform implementation may itself vary in terms of availability of platform services.
- *Domain-Specific services:* an INDENICA platform includes also domain-specific services that extend beyond the technical level. These can vary.
- *Service composition (and processes):* even within the INDENICA-platform services may be composed. This is often done by integrating them in the form of a process, that itself can be seen as a service. Variation may occur in the constituents of the composition.
- *Service and platform deployment:* services or a complete platform is deployed does not impact the logical content of a platform, but it strongly impacts the physical layout and thus many characteristics of the running platform. Deployment can be an object of variation as well.

Form of Variation: We can differentiate multiple forms of variation. Most of them are derived from variability management in general, but some are inspired by service-orientation:

- Optional: a variability object may only be part of an installation under certain circumstances. This is called Optional Variability.
- Alternative: sometimes it is important that one of several variability objects is present, but the variation is in which of the objects to pick.
- Multiple selection: sometimes multiple options from a set of variability objects can be selected.
- Parameterization: variation can be communicated through a parameter, respectively the value.
- Extension: variation can occur by extending a feature, service or property that already exists. Typically, this is done by providing explicit extension points.
- Interface: the interface (e.g., of a service) is adapted, e.g. by modifying number and type of its parameters.

Binding Time: This determines when a choice of the specific variation needs to be made. Typical examples are implementation time (explicit human interference is required, e.g., to modify a program description), compile time, deployment time, initialization time, service-binding time, and runtime (meaning at any time during the operation of the system). Further a binding of variation can either be permanent or volatile (in the later case a rebinding for a new variation is possible). Some techniques also support multiple binding stages; this will be noted as well.

Context: This section describes the context in which the technique can be used. While this does not mean that the technique cannot be used outside of this context, it is at least not so straightforward or not based on existing experience.

Environment Contexts: This describes the technical dependencies to the environment the approach has. For example a specific technique might be proven with Web Services or in the OSGi context. This does not mean it cannot be applied outside the context, but only that no experience its application exist so far.

Assumptions on Systems: Techniques may make further assumptions on the types of the systems they will be applied in. If applicable, they will be listed here.

Solution: This section describes the technical solution. While we expect that several techniques will be available that have the same or similar purpose, we will only describe unique solutions as independent patterns.

Key idea: This describes the key idea of the technique, we found. We will aim to transfer the description into the terminology used throughout this deliverable, as far as possible.

Technology Background: If specific implementation technologies (e.g., aspect-oriented techniques) are required for using this technique, we list them and characterize their role, with respect to the variability implementation.

Variability Approach: This characterizes the assumptions the technique makes about the relation of the variable parts and the core parts of the implementation. A technique may employ positive or negative variability: positive means variable parts are added to the core implementation (aspect-oriented approaches are typically an example of this), negative means all variable parts are part of the main implementation and not applicable variation is removed. In generation-based variability the necessary implementation is generated from a description in a different form. If a categorization on this basis is not possible, we will put it as *not applicable*.

Variability Granularity and Selection: This describes the level of granularity on which variability can be described and how individual variable elements can be selected for inclusion or exclusion into a variant realization. Typical examples are line of code, file, statement, parameter, function, module, service, service binding, service bundle, deployment unit. Effective selection typically happens via some sort of mark-up or reference but may also be handled externally.

Dependency Management Support: If the technique provides direct support for management of dependencies among variant selection, this is described here.

Platform Definition Support: If the technique also supports the identification and definition of the variable parts that are needed to make up a service platform, this is described here.

Further Aspects: Further information regarding the technique that does not fit the above categories is described here.

Source: The sources for the information described above are listed here. This may include literature or company references.

Comments: Any information that does not fit the categories above, but is considered important for the purpose of the analysis is described here.

Using the above pattern template, we analysed the techniques that aimed at product line variability and described them in this way. We aimed at being exhaustive in this effort, thus also some borderline cases are included that do not provide a well-defined or unique technique. The overview is organized according to the variability objects we defined above to achieve good accessibility of the results from a problem perspective. Out of the different variability objects, we did not address *service platform infrastructure* as this can be implemented using any kind of variability and providing a survey on this would have led to replicating existing, generic surveys of product line implementation such as [53, 78].

A.1 Variability in Service Composition and Processes

A.1.1 Service Composition Generation

Name: Service composition generation

Purpose:

Description: Support variability of service composition by generating a specific business process (in BPEL [63]) based on feature configuration and the selection of corresponding domain activities of a Business Process Family Model (BPFM). A BPFM specifies common and variable services in the same way a Feature Model specifies common and variable features in traditional product line engineering.

Variability Object: Service composition (and processes)

Form of Variation: Alternative, Optional

Binding time: Compile time, permanent

Context:

Environment Contexts: The existing implementation requires Web Service technology and the use of UML 2.0 activity diagrams as a basis for BPFM modelling.

Assumption on Systems: -

Solution:

Key idea: Generation of specific business processes (in BPEL) from a generic BPFM including common and variable services. The BPFM is modelled using an extension of UML activity diagrams including variation points, variation point bindings and variants regions (introduced by this approach). The generation of a business process variants from the BPFM is based on a given feature configuration and the selection of the corresponding core domain activities and variants (used to implement the selected features). This is, deriving a UML activity diagram in which all variability is resolved. The automated transformation of an UML 2.0 activity diagram [58] to BPEL is presented in [15, 38, 21].

Technology Background: Transformation component is required to automatically transform a business process variant derived from the BPFM (UML activity diagram) into a business process defined in BPEL.

Variability Approach: generation-based

Variability Granularity and Selection: Service

Dependency Management Support: Rules and dependencies among services are managed on the level of the feature model by specifying relations and dependencies among features using the FODA methodology [42].

Platform Definition Support: -

Further Aspects:

Source: [64, 52] and the conceptually equal Business Process Line (BPL) approach proposed in [14]

Comments: -

A.1.2 Component Service Replacement

Name: Component service replacement

Purpose:

Description: Support variability of service composition by replacing one or more component services of a business process. A component service provides functionality that is required to achieve the overall goal of the business process. Variability exists in the way that component services are replaced to meet varying QoS constraints.

Variability Object: Service composition (and processes)

Form of Variation: Alternative

Binding time: Runtime, volatile

Context:

Environment Contexts: The existing implementation requires Web Service technology.

Assumption on Systems: -

Solution:

Key idea: Triggered by a QoS constraint violation, each component service of the business process is analysed to identify one or more component services that have to be replaced in order to satisfy the QoS constraint (this is based on a set of QoS attributes, their values and calculation algorithms). The replacement of component services does not affect the overall structure of the business process.

Technology Background: -

Variability Approach: *not applicable*

Variability Granularity and Selection: Service

Dependency Management Support: Rules and dependencies among services only exist in the sense of QoS attributes and constraints on the overall QoS value of the business process. No specific variability modelling approach is given.

Platform Definition Support: -

Further Aspects:

Source: [46]

Comments: -

A.1.3 Scoping and Fine-tuning

Name: Scoping and fine-tuning

Purpose:

Description: Adaptation of a generic service platform to specific business needs.

Variability Object: Domain-specific Services

Form of Variation: Optional, Alternative, Multiple selection, Parameterization

Binding time: Runtime, volatile

Context:

Environment Contexts: Scoping and fine-tuning is a solution for Business By Design, SAPs on-demand platform.

Assumptions on Systems: -

Solution:

Key idea: SAP provides standard business solutions targeting several fields of industries as well as companies with different preconditions and requirements. So a single company must be able to select only the functionality needed for the specific business needs. Additional to the Business By Design platform SAP delivers at one hand a comprehensive catalogue (BAC, Business Adaptation Catalogue) exposing the entire set of solution capabilities, described using non-technical business language. At the other hand SAP and partners deliver so called BC sets which contains predefined business configurations. At first the customer has to make selection from the BAC based on his specific business needs (scoping), at second he can overwrite parameters of the predefined configuration (fine-tuning). The results are stored in a configuration workspace. During all this steps constraints will be checked, to receive a consistent solution. The final configuration becomes active through Business Configuration (BC) deployment. The process comprises activation of UI components and services as well as writing the configuration to Customizing Tables, which are evaluated at runtime.

Technology Background: -

Variability Approach: negative variability

Variability Granularity and Selection: Service, external description

Dependency Management Support: Constraint checks to receive a consistent solution.

Platform Definition Support: -

Further Aspects:

Source: SAP[12]

Comments:

A.2 Variability in Domain-Specific Services

A.2.1 Component-Based Service Implementation

Name: Component-Based Service implementation

Purpose:

Description: The approach is rather generic and more a conceptual framework than a single approach. It adds a component layer as refinement of services and mainly aims at realizing variability within the component level. The approach also supports and allows variability on a higher level like a business process. There is no single implementation technique prescribed, however, a focus on component-based implementation techniques to realize variable service functionality (i.e., different component compositions, pre-processors, aspect-orientation).

Variability Object: Domain-specific services

Form of Variation: Alternative, Optional

Binding time: Compile time, permanent

Context:

Environment Contexts: Service technology is required but the approach does not focus on any specific service technology. Component-based development (CBD) is used to implement each variant (feature) in a different component.

Assumption on Systems: -

Solution:

Key idea: Each component represents a specific variant (feature). Based on a feature selection, the corresponding components are composed to provide the desired service functionality. Superfluous components (features that are not selected) can be removed.

Technology Background: -

Variability Approach: s*upposedly can be combined with negative and positive approaches.*

Variability Granularity and Selection: Module, Service

Dependency Management Support: Rules and dependencies among components are managed on the level of the feature model. In the feature

[12]http://help.sap.com/saphelp_byd30/en/KTP/Software-Components/01200615320100003379/SAP_BBD/SAP_BBD.html

model relations and dependencies among the features can be defined. No specific feature modelling approach is given.

Platform Definition Support: -

Further Aspects:

Source: [48]

Comments: -

A.2.2 Pattern Plugin

Name: Pattern Plugin

Purpose:

Description: The approach uses model-based techniques to derive a specific implementation using a business process with variabilities as a basis. The variation point information is then used to derive a specific implementation by repeatedly applying some patterns at the variation point.

Variability Object: Domain-specific services, service-composition and processes

Form of Variation: Alternative, Optional, Interface

Binding time: Compile time, permanent

Context:

Environment Contexts: Service technology is required but the approach does not focus on any specific technology.

Assumption on Systems: -

Solution:

Key idea: A SOA solution is modelled based on the meta-model of the approach (results in a design model) and is augmented with variation points. Each variant of a variation point has its own variation model that holds the information about added, removed or modified implementations, interfaces or processes. A new solution is defined by choosing the appropriate variations. The variation models of the desired variants are composed into the primary design model, via pattern application. The composed model can finally be transformed into code artefacts.

Technology Background: UML modelling tool to define the pattern plugins which are applications that impose design patterns on an existing design model.

Variability Approach: generation-based

Variability Granularity and Selection: variables, operations, interface parts

Dependency Management Support: Rules and dependencies between the artefacts are modelled separately in the form of a Constraint Satisfaction

Problem (CSP); constraint checker are used to check for consistency among the constraints.

Platform Definition Support: -

Further Aspects:

Source: [54]

Comments: -

A.2.3 FOP-based Refinement

Name: FOP-based refinement

Purpose:

Description: Variability of service implementation and service interfaces can be supported by applying feature-oriented programming (FOP). In such an approach the code of a feature is encapsulated into a feature module and used to the refine the service's base code by joining the base code and the code of a feature. Actual implementation in this approach is done by delegating the work to implementation techniques like aspect-orientation or composition based on the AHEAD tool suite [11].

Variability Object: Domain-specific services

Form of Variation: Optional, Alternative, Extension, Interface

Binding time: Compile time, permanent

Context:

Environment Contexts: The current implementation requires Web Service technology as WSDL is used to define interfaces and interface refinements.

Assumption on Systems: -

Solution:

Key idea: Features are separated from the basic service architecture by implementing them as individual feature modules. Each feature module consists of one or more extensions (e.g. a set of Java class refinements and WSDL interface refinements) to the artefacts of the base services the feature affects. If a feature is selected at configuration time, the extensions are applied to the affected service base classes and interface (feature composition). To extend the correct classes and interfaces, the extensions have the same name than the base implementation or interface.

Technology Background: Extensions to programming languages to support features, e.g. Java [11], C++ [7], XML [5], and WSDL [8]. The extensions are required for feature modularization and composition.

Variability Approach: Positive variability

Variability Granularity and Selection: Statement, Function, Module

Dependency Management Support: In this approach, FOPS performs feature composition based on a declarative specification [11]

Platform Definition Support: -

Further Aspects:

Source: [6]

Comments: Service implementation has to be accessible (single vendor, white-box-service), or a common feature model must exist that allows vendors to provide a feature-based specification of their services (multiple vendors, black-box-service).

A.2.4 Class Wrapper

Name: Class wrapper

Purpose:

Description: Support variability of service implementations at runtime by applying feature-oriented programming (FOP) techniques to encapsulate the code of a feature into a feature module and refine service's base code by joining the base code and the code of a feature via Java HotSwap at runtime. Java HotSwap is incorporated within the debugger API (since Java 1.4 JVM) and allows debuggers to update bytecode in place using the same class identity [41].

Variability Object: Domain-specific services

Form of Variation: Extension (including Alternative and Optional)

Binding time: Runtime, volatile

Context:

Environment Contexts:-

Assumption on Systems: -

Solution:

Key idea: Features are separated from the basic service architecture by implementing them as individual feature modules. Each feature module consists of one or more classes and wrappers. Adding a feature at runtime requires loading all base classes and wrappers it introduces. Base class code updates only internal algorithms without affecting the class schema. Wrappers are used to apply new elements such as new methods to a class. In order to invoke the new or modified functionality provided by the wrapper all object references of the changed class have to be updated. This is also achieved via method reimplementation based on Java HotSwap.

Technology Background: The implementation requires Java HotSwap, which is a feature of Oracles standard Java virtual machine called HotSpot [61]. HotSwap is required to add features in terms of base classes and wrappers to the implementation at runtime.

Variability Approach: Positive variability

Variability Granularity and Selection: Function, Module, Service

Dependency Management Support: Rules and dependencies among features are managed on the level of the feature model. No specific feature modelling approach is expected.

Platform Definition Support: -

Further Aspects:

Source: [75]

Comments: Similar to A.2.3, a service implementation has to be accessible to be able to apply features.

A.2.5 Aspect Service Weaver

Name: Aspect Service Weaver

Purpose:

Description: Support variability of service implementation by weaving the code of advice services into existing services based on requests for certain functionality. An advice service implements additional code (representing the variability that can be woven into existing services). Weaving is done by a tool called Aspect Service Weaver (ASW) [10, 40, 81] that is introduced by this approach. Additional functionality can be requested at configuration time as well as at runtime.

Variability Object: Domain-specific services

Form of Variation: Optional, Alternative, Extension

Binding time: Runtime (as explained in the approach, but configuration time is also possible), permanent

Context:

Environment Contexts: The existing implementation requires Web Service technology supporting message transaction based on SOAP protocol as ASW only intercepts SOAP messages at runtime.

Assumption on Systems:-

Solution:

Key idea: ASW intercepts the SOAP messages between service consumer and provider at runtime. If such a message includes a request for a method that the service does not support currently, advice services implementing the variability are queried. These advice services implement the additional methods that can be woven into existing services by the ASW. Thus, dynamically requested methods are implemented by advices and deployed at runtime by weaving the corresponding advices into the existing service.

Technology Background: AOP technology required

Variability Approach: Positive variability

Variability Granularity and Selection: Function, Module

Dependency Management Support: -

Platform Definition Support: -

Further Aspects:

Source: [51]

Comments: Limitation to the capabilities of the ASW and the AOP language; further limitation to SOAP message interceptions. Compile time weaving is considered as variability implementation technique for traditional software product lines [53].

A.2.6 Enhancement Options

Name: Enhancement Options

Purpose:

Description: Enrich services with additional functionality or overwrite existing functionality to provide variability for services at runtime without modifying the original implementation.

Variability Object: Domain-specific Services

Form of Variation: Extension

Binding time: Runtime, volatile

Context:

Environment Contexts: The Enhancement and Switch Framework is a solution to enhance business functionality of SAPs NetWeaver ABAP core.

Assumptions on Systems: -

Solution:

Key idea: SAP provides standard business solutions, which are generic in nature and cannot cover all specific business needs. There is a need to combine the advantages of both, the standard (easily maintainable) with the proprietary (more flexible) solutions while avoiding the drawbacks of standard software (lack of flexibility) and customized software (upgrade issues). The Enhancement Framework allows this by defining Enhancement Options that are hooks in the original code, which bind the modifications during runtime. Enhancement Options can be implicit or explicit. Implicit Options are provided by the framework (e.g. begin of methods), explicit options are defined by the developer. There are two kinds of explicit options - source code plug-ins and object plug-ins. With source code plug-ins the developer can define Enhancement Points, which are hooks for adding additional code or Enhancement Sections, which are hooks for overwriting existing code. An object plug-in is called BAdi (Business Add-in). It comprises an interface defining methods, which add new and changeable functionality to an object.

The enhancement options can be activated and deactivated with the help of the Switch Framework.

Technology Background: Aspect-orientation-

Variability Approach: positive variability

Variability Granularity and Selection: line of code

Dependency Management Support: -

Platform Definition Support: -

Further Aspects:

Source: SAP[13]

Comments:

A.3 Variability in Service and Platform Deployment

A.3.1 Generation of Deployment / Undeployment Scripts

Name: Generation of Deployment / undeployment scripts

Purpose:

Description: Support variability of the deployment of services by generating individual deployment (and undeployment) scripts based on infrastructure configuration.

Variability Object: Service and platform deployment

Form of Variation: Optional, Alternative

Binding time: deployment time, runtime, volatile

Context:

Environment Contexts: The existing implementation is based on Web Service technology. A suitable language to describe and execute deployment scripts is Web Service Business Process Execution Language WS-BPEL [63] as shown in [44, 49, 69]. The current example is based on the Apache Tuscany service component architecture (SCA) runtime.

Assumption on Systems:-

Solution:

Key idea: Annotation of the variability model with deployment information for each variant. Based on a configuration the planning component of the deployment infrastructure can generate a specific deployment script by composing the deployment information of each selected variant.

Technology Background: The variability implementation technique requires a deployment infrastructure that is capable of executing the WS-BPEL deployment scripts.

[13] http://help.sap.com/saphelp_nw73/helpdata/en/

Variability Approach: Positive variability

Variability Granularity and Selection: Service Binding, Deployment Unit

Dependency Management Support: Rules and dependencies among variants are managed on the level of the variability model. The Orthogonal Variability Modelling (OVM) approach for variability management is used as an example.

Platform Definition Support: -

Further Aspects:

Source: [50]

Comments: -

A.3.2 Context-aware Deployment Plan

Name: Context-aware deployment plan

Purpose:

Description: Support variability of the deployment of components to different execution locations at runtime by replacement of component instances including state transfer and rewired inter-component connections.

Variability Object: Service and platform deployment

Form of Variation: Optional, Alternative, volatile

Binding time: deployment time, runtime

Context:

Environment Contexts: The existing implementation requires a CORBA runtime environment supporting the CORBA Component Model (CCM) deployment model [55]. In the platform, further management components are required to provide a safe redeployment.

Assumptions on Systems: -

Solution:

Key idea: Extension of the deployment model by including (architecture) variation points to affect the distribution of components and the deployed component implementation versions (implementation variants). The so-called context-aware deployment plan contains the initial deployment and the supported variability at runtime. Runtime variability of the deployment plan is handled by generating new deployment plans in terms of removing or adding components, removing or adding component connections as well as selecting a component due to its implementation version. In the specific approach a consistency manager enforces specific component states for redeployment. Furthermore, a consistency manager blocks new connections during redeployment, handles the state transfer among components as well the reestablishment of component connections to the replaced component instance.

Technology Background: -

Variability Approach: generation-based

Variability Granularity and Selection: service, service binding

Dependency Management Support: Constraints may be specified on component connections for expressing required components or component connections.

Platform Definition Support: -

Further Aspects:

Source: [9]

Comments: -

A.4 Variability in Technical Platform Services

A.4.1 Abstract Roles

Name: Abstract Roles

Purpose:

Description: Integrate services from different middleware implementations into one application specific middleware using runtime variability to dynamically adapt service compositions and service interactions to meet a given Quality of Service (QoS).

Variability Object: Platform service, Domain specific services

Form of Variation: Extension (including Alternative and Optional)

Binding time: Initialization time, Runtime, volatile

Context:

Environment Contexts: In the concrete realization, the application-specific middleware is an instance of the generic ROAD library (Role-oriented Adaptive Design). The integration of middleware platforms relies on WSDL-SOAP.

Assumptions on Systems: Due to the WSDL-SOAP protocol overhead, the realization is (probably) limited to systems, which do not require high load or small response times.

Solution:

Key idea: The (aggregated) application-specific middleware performs no domain-specific function by itself but provides abstract functional roles that can dynamically be played by other entities, e.g. services provided by middleware implementations to be integrated. Adapters realize the roles for concrete middleware implementations and brokers support the flexible communication among adapters. The instantiation of concrete adapters in the aggregated middleware realizes binding of the variabilities and the integration of existing functionality.

Technology Background: The approach relies on association aspects, an extension of AspectJ [1]. Association aspects are used to restrict the communication among roles by message interception (c.f. Section A.4.3).

Variability Approach: positive variability

Variability Granularity and Selection: module, service, service binding

Dependency Management Support: Contracts define the topology among roles in terms of provided, required and permissible communication as well as non-functional (QoS) requirements.

Platform Definition Support: -

Further Aspects:

Source: [23]

Comments:-

A.4.2 Application-specific Callbacks

Name: Application-specific callbacks

Purpose:

Description: Supply application-specific parts to existing middleware services. Middleware services call the domain-specific implementation via (remote) callbacks.

Variability Object: Platform service

Form of Variation: Extension (is also used to realize optional and alternative)

Binding time: Runtime, permanent (during the lifetime of a transaction)

Context:

Environment Contexts: The existing implementation targets application-specific callbacks in web-based systems (realized for JBoss/Struts, CORBA and .NET), i.e. callbacks between server and browser-based clients using the HTTP protocol.

Assumptions on Systems: -

Solution:

Key idea: Existing platform services provide callback interfaces so that domain-specific functionality can be executed at certain points of service execution. New components are defined in an extra layer on top of the platform and register themselves as callback handlers. Variabilities are bound when registering the callback handlers.

Technology Background: -

Variability Approach: positive variability

Variability Granularity and Selection: function

Dependency Management Support: -

Platform Definition Support: -

Further Aspects:

Source: [32, 31]

Comments:

A.4.3 Extension by Interception

Name: Extension by Interception

Purpose:

Description: Modify or replace management services in an existing middleware platform or a service container by dynamically manipulating call chains.

Variability Object: Platform service, Domain-specific service

Form of Variation: Extension (including Optional, Alternative)

Binding time: Initialization Time, Runtime, volatile

Context:

Environment Contexts: The platform needs to provide a specific call chain for each distinct management service. The existing implementation uses the JBoss Application Server as underlying platform.

Assumptions on Systems: -

Solution:

Key idea: Instead of direct calling a callee from a caller the call is represented as a call object and passed through a chain of interceptors. Each interceptor may work with the data, modify, consume or reissue the call object. For each type of call the platform must provide appropriate interceptor chains. Custom interceptors may handle calls in different, domain-specific ways. Variability is bound by adding, replacing or removing interceptors.

Technology Background: -

Variability Approach: typically positive variability, negative variability can be realized by removing or replacing interceptors from a default interceptor chain

Variability Granularity and Selection: function, module

Dependency Management Support: -

Platform Definition Support: -

Further Aspects:

Source: [31, 32]

Comments: Interceptor chains in web or service containers are typically only traversed if the invocation comes from a call to a (service) interface through

the (service) container. Calls among (service) interfaces are typically not subject to interception chains.

A.4.4 Aspect-oriented Composition

Name: Aspect-oriented composition

Purpose:

Description: Introduce variability into existing platforms, which provide certain functionality. Reuse the platforms without modifying their code base.

Variability Object: Platform service, Domain-specific service

Form of Variation: Extension (including Optional, Alternative)

Binding time: Compile time, Initialization time, Runtime, permanent

Context:

Environment Contexts: The existing implementation applies the technique to a CORBA Object Request Broker realized in component-based design in order to support dynamic substitution of objects during marshalling and unmarshalling or to add additional interceptors for domain specific functionality (see Section A.4.3)

Assumptions on Systems: -

Solution:

Key idea: A core platform, which exhibits defined extension points is used as common infrastructure. Variabilities (or their binding to the core) are expressed as aspects realizing certain extension points. Variabilities are bound by weaving the aspects into the core platform.

Technology Background: An aspect weaver is required at the respective binding time. In [84, 83] the utilized aspect technology is enhanced by an aspect-based variability model, which enriches the static and dynamic weaving functionality and ensures thread-safe aspect oriented composition.

Variability Approach: positive variability

Variability Granularity and Selection: function, module

Dependency Management Support: The aspect-based variability model supports depends and conflicts relationships among variabilities.

Platform Definition Support: -

Further Aspects:

Source: [84, 83]

Comments: AOP techniques are considered as a general variability technique on various levels of granularity [53]. Due to required compatibility with the type system of the underlying programming language, AOP may particularly not remove existing methods, interfaces or types and, thus, may typically realize only additive variability. The authors of [36] aim at balancing the

overhead by domain-specific resource-aware weavers, i.e. the weaver itself is considered as subject to (dynamic) variability.

A.4.5 Reflective Variability, Meta-data-based Variability

Name: Reflective variability, meta-data based variability

Purpose:

Description: Use meta-information on the system and its constituting components (such as topology of components, component interfaces, etc.) in order to determine and reason about variation points and available variants at runtime.

Variability Object: Platform Service, Domain-specific Service

Form of Variation: Extension

Binding time: Initialization time, Runtime, volatile

Context:

Environment Contexts: The existing implementation is based on the domain-specific language tool Genie [13], which is used as a basis to generate software artefacts, component framework configurations and reconfiguration policies.

Assumptions on Systems: -

Solution:

Key idea: Reflective capabilities of a component framework offer the potential to reason about the possible variation points and their variants at runtime. Reflection is the inspection and manipulation of causally connected meta-models of a software system as exposed via their meta-interface. Typical meta-models cover the functional component interface, the architecture (topology of components) or interception support (execution of component functionality via the interface meta-model). Reflective techniques can be applied to realize (runtime) variability, i.e. to gain information on the running system such as the concrete locations of variation points or applicable variations as well as to manipulate the system by binding concrete variabilities.

Technology Background: A middleware with reflective capabilities such as GridKit [13] is required. A programming language with built-in reflection capabilities may simplify the realization.

Variability Approach: negative variability

Variability Granularity and Selection: function, module, service, service bindings

Dependency Management Support: -

Platform Definition Support: -

Further Aspects:

Source: [12]

Comments: An interception meta-model allows runtime modification to existing interceptor chains (c.f. Section A.4.3). In [24] an extensible set of orthogonal meta-models can be attached or unattached from components on demand at runtime.

A.4.6 Event-based Composition, Publish/Subscribe-Composition

Name: Event-based composition, Publish/Subscribe-Composition

Purpose:

Description: Add or remove management services from a middleware platform and select the correct (concrete) implementation of individual management services based on functional and non-functional characteristics.

Variability Object: Platform service

Form of Variation: Alternative, Optional

Binding time: Compile time, permanent

Context:

Environment Contexts: The described implementation is based on the FamiWare microkernel [33, 34].

Assumptions on Systems: The approach targets Ambient Intelligence Systems.

Solution:

Key idea: A middleware platform provides a core set of event services. The event services deliver communication events among subscribed platforms or domain-specific application services. Services are realized as pluggable components, which subscribe to relevant event services in order to react on or modify received events. Variability can be obtained by unregistering existing services and registering new services.

Technology Background: -

Variability Approach: positive variability

Variability Granularity and Selection: service, service binding

Dependency Management Support: Logical expressions which can be reduced to usage and mutual exclusion.

Platform Definition Support: -

Further Aspects:

Source: [33, 34]

Comments:

A.4.7 Generated Component Connectors

Name: Generated Component Connectors

Purpose:

Description: Obtain a domain-specific deployment tool for a component-based platform.

Variability Object: Platform service

Form of Variation: Extension

Binding time: Compilation time, permanent

Context:

Environment Contexts: The described implementation relies on the SOFA2 [19] component system.

Assumptions on Systems: -

Solution:

Key idea: Entities realizing the concrete communication among components are encapsulated as so called connectors. The generic deployment tool itself is described as a skeleton. The skeleton is filled by generating the code of the instantiations for the concrete connectors selected in the product line configuration.

Technology Background: -

Variability Approach: negative variability

Variability Granularity and Selection: module

Dependency Management Support: -

Platform Definition Support: Allows deriving domain-specific deployment tools.

Further Aspects:

Source: [18]

Comments:

A.4.8 Microcomponents

Name: Microcomponents

Purpose:

Description: Obtain a domain-specific execution environment by varying the lifecycle control of the components.

Variability Object: Platform service

Form of Variation: Extension

Binding time: Initialization time, Runtime, volatile

Context:

Environment Contexts: The described implementation is realized as part of the SOFA2 [19] component system.

Assumptions on Systems: -

Solution:

Key idea: Represent individual control functionalities in the lifecycle of a component such as starting a component, interface lookup etc. as simple subcomponents called microcomponents. Microcomponents can be grouped into “aspects” in order to define consistent extensions of the lifecycle control mechanisms. Variation is bound when adding, replacing or removing microcomponents or aspects from the lifecycle control of a platform.

Technology Background: -

Variability Approach: positive variability

Variability Granularity and Selection: module, service

Dependency Management Support: grouping of microcomponents

Platform Definition Support: -

Further Aspects:

Source: [18]

Comments:

A.4.9 Use Platform Management Services

Name: Use platform management services

Purpose:

Description: Use the management services of a platform to obtain a loosely coupled component or service-based platform to enable variability.

Variability Object: Platform service, Domain-Specific services, Service composition

Form of Variation: Extension

Binding time: Service-binding time, Runtime, volatile

Context:

Environment Contexts: Component or service-based systems.

Assumptions on Systems: -

Solution:

Key idea: Realize variable functionality in terms of the component or service platform and use the provided management functionality for suspending, resuming, deploying, adding, binding or deleting components or services. This allows decoupling variabilities and supports runtime variability and runtime (service) composition. Therefore modularization takes places on two axes.

- Deployment units (e.g. bundles in OSGi) have well-defined interfaces and dependencies. The linking of the deployment units happens during

startup of the component. Deploying and selecting specific bundles enables service implementation variability.

- Deployment units can export and import services. Registration and deregistration of services occurs at runtime. This allows (service) composition variability at runtime as services are registered with a symbolic name.

Technology Background: Depends on the capabilities of the underlying component or service platform, e.g. OSGi [79] at SAP or SCA in [65, 66]

Variability Approach: positive variability

Variability Granularity and Selection: module, service

Dependency Management Support: -

Platform Definition Support: Automatic resolution of dependencies between deployment units may happen based on requirements, offerings and optionally further functional or non-functional constraints.

Further Aspects:

Source: [65,66], SAP

Comments:

References

[1] Project homepage AspectJ, 2011. Online available at: http://www.eclipse.org/aspectj/.

[2] Project homepage Mobicents, 2011. Online available at: http://www.mobicents.org/.

[3] Project homepage Pococapsule, 2011. Online available at: http://code.google.com/p/pococapsule/.

[4] Project homepage Virgo, 2011. Online available at: http://www.eclipse.org/virgo/.

[5] F. I. Anfurrutia, O. Díaz, and S. Trujillo. On Refining XML Artifacts. In *Proceedings of the 7th International Conference on Web Engineering (ICWE '07)*, pages 473–478, 2007.

[6] S. Apel, C. Kaestner, and C. Lengauer. Research Challenges in the Tension Between Features and Services. In *Proceedings of the 2nd International Workshop on System Development in SOA Environments (SDSOA '08)*, pages 53–58, 2008.

[7] S. Apel, T. Leich, M. Rosenmueller, and G. Saake. FeatureC++: On the Symbiosis of Feature-Oriented and Aspect-Oriented Programming. In *Proceedings of the 4th International Conference on Generative Programming and Component Engineering (GPCE '05)*, pages 125–140, 2005.

[8] S. Apel and C. Lengauer. Superimposition: A Language-Independent Approach to Software Composition. In *Proceedings of the 7th International Conference on Software Composition (SC '08)*, pages 20–35, 2008.

[9] D. Ayed and Y. Berbers. Dynamic Adaptation of CORBA Component-Based Applications. In *Proceedings of the 22nd Annual ACM Symposium on Applied Computing (SAC '07)*, pages 580–585, 2007.

[10] F. Baligand and V. Monfort. A Concrete Solution for Web Services Adaptability Using Policies and Aspects. In *Proceedings of the 2nd International Conference on Service Oriented Computing (ICSOC '04)*, pages 134–142, 2004.

[11] D. Batory, J. N. Sarvela, and A. Rauschmayer. Scaling Step-Wise Refinement. In *Proceedings of the 25th International Conference on Software Engineering (ICSE '03)*, pages 187–197, 2003.

[12] N. Bencomo, G. Blair, G. Coulson, and T. Batista. Reflective Component-Based Technologies to Support Dynamic Variability. In *Proceedings of the 2nd International Workshop on Variability Modelling of Software-intensive Systems (VAMOS '08)*, 2008.

[13] N. Bencomo, G. Blair, and P. Grace. Models, Reflective Mechanisms and Family-Based Systems to Support Dynamic Configuration. In *Proceedings of the 1st Workshop on Model Driven Development for Middleware (MODDM '06)*, 2006.

[14] N. Boffoli, M. Cimitile, F. M Maggi, and G. Visaggio. Managing SOA System Variation through Business Process Lines and Process Oriented Development. In *Proceedings of the 3rd Workshop on Service-Oriented Architectures and Software Product Lines: Enhancing Variation (SOAPL '09)*, 2009.

[15] B. Bordbar and A. Staikopoulos. On Behavioural Model Transformation in Web Services. In S. Wang, K. Tanaka, S. Zhou, T.-W. Ling, J. Guan, D Yang, F. Grandi, E. Mangina, I.-Y. Song, and H. Mayr, editors, *Conceptual Modeling for Advanced Application Domains*, volume 3289 of *Lecture Notes in Computer Science*, pages 667–678. Springer Berlin / Heidelberg, 2004. 10.1007/978-3-540-30466-1_61.

[16] H. P. Breivold and M. Larsson. Component-Based and Service-Oriented Software Engineering: Key Concepts and Principles. In *Proceedings of the 33rd EUROMICRO Conference on Software Engineering and Advanced Application (EUROMICRO-SEAA '07)*, pages 13–20, 2007.

[17] T. Bures. *Generating Connectors for Homogeneous and Heterogeneous Deployment*. PhD thesis, Charles University in Prague, Faculty of Mathematics and Physics, 2006.

[18] T. Bures, P. Hnetynka, and M. Malohlava. Using a Product Line for Creating Component Systems. In *Proceedings of the 24th Annual ACM Symposium on Applied Computing (SAC '09)*, pages 501–508, 2009.

[19] T. Bures, P. Hnetynka, and F. Plasil. SOFA 2.0: Balancing Advanced Features in a Hierarchical Component Model. In *Proceedings of the 4th International Conference on Software Engineering Research, Management and Applications (SERA '06)*, pages 40–48, 2006.

[20] F. Buschmann, K. Henney, and D. C. Schmidt. *Pattern-oriented Software Architecture: On Patterns and Pattern Languages*. John Wiley and Sons, 2007.

[21] J. Bézivin, S. Hammoudi, D. Lopes, and F. Jouault. Applying MDA Approach to B2B Applications: A Road Map. In *Proceedings of the 2004 Workshop on Model Driven Development (WMDD '04)*, 2004.

[22] L. Cavallaro, E. Di Nitto, P. Pelliccione, M. Pradella, and M. Tivoli. Synthesizing Adapters for Conversational Web-Services from their WSDL Interface. In *Proceedings of the 5th ICSE International Workshop on Software Engineering for Adaptive and Self-Managing Systems (SEAMS '10)*, pages 104–113, 2010.

[23] A. Colman, L. D. Pham, J. Han, and J.-G. Schneider. Adaptive Application-Specific Middleware. In *Proceedings of the 1st Workshop on Middleware for Service Oriented Computing (MW4SOC '06)*, pages 6–11, 2006.

[24] G. Coulson, G. Blair, P. Grace, F. Taiani, A. Joolia, K. Lee, J. Ueyama, and T. Sivaharan. A Generic Component Model for Building Systems Software. *ACM Transaction on Computer Systems*, 26:1:1–1:42, March 2008.

[25] I. Crnkovic and H. P. Breivold. Tutorial: Emerging Technologies in Industrial Context - Component-Based and Service-Oriented Software Engineering. In *Proceedings of the 31st IEEE International Computer Software and Applications Conference (COMPSAC '07)*, 2007.

[26] I. Crnkovic, S. Sentilles, A. Vulgarakis, and M. R.V. Chaudron. A Classification Framework for Software Component Models. *IEEE Transactions on Software Engineering*, 99(PrePrints), 2010.

[27] K. Czarnecki, S. Helsen, and U. Eisenecker. Formalizing Cardinality-based Feature Models and their Specialization. *Software Process: Improvement and Practice, Special Issue on Software Variability: Process and Management*, 10(1):7 – 29, 2005.

[28] E. Dolstra, G. Florijn, M. de Jonge, and E. Visser. Capturing Timeline Variability with Transparent Configuration Environments. In *Proceedings of the 1st International Workshop on Software Variability Management (SVM '03)*, 2003.

[29] E. Dolstra, G. Florijn, and E. Visser. Timeline Variability: The Variability of Binding Time of Variation. In J. van Gurp and J. Bosch, editors, *Proceedings of the 1st International Workshop on Software Variability Management (SVM '03)*, 2003.

[30] Organization for the Advancement of Structured Information Standards. UDDI Spec Technical Committee Draft, Version 3.0.2, 2004. Online available at: http://www.uddi.org/pubs/uddi_v3.htm.

[31] L. Froihofer, K. M. Goeschka, and J. Osrael. Middleware Support for Adaptive Dependability. In *Proceedings of the 8th ACM/IFIP/USENIX International Conference on Middleware (Middleware '07)*, pages 308–327, 2007.

[32] L. Froihofer, J. Osrael, and K. M. Goeschka. Middleware/Application Interactions to Support Adaptive Dependability. In *Proceedings of the 1st Workshop on Middleware-Application Interaction (MAI '07)*, pages 31–36, 2007.

[33] L. Fuentes and N. Gámez. Configuration Process of a Software Product Line for AmI Middleware. *Journal of Universal Computer Science*, 16(12):1592–1611, 2010.

[34] N. Gámez and L. Fuentes. FamiWare: a family of event-based middleware for ambient intelligence. *Personal Ubiquitous Computer*, 15:329–339, April 2011.

[35] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns. Elements of Reusable Object-Oriented Software*. Addison-Wesley, Massachusetts, 2000.

[36] W. Gilani, N. H. Naqvi, and O. Spinczyk. On Adaptable Middleware Product Lines. In *Proceedings of the 3rd Workshop on Adaptive and Reflective Middleware (ARM '04)*, pages 207–213, 2004.

[37] I. Groher and M. Voelter. XWeave: Models and Aspects in Concert. In *Proceedings of the 10th International Workshop on Aspect-Oriented Modeling (AOM '07)*, pages 35–40, 2007.

[38] R. Gronmo and M. C. Jaeger. Model-Driven Semantic Web Service Composition. In *Proceedings of the 12th Asia-Pacific Software Engineering Conference (APSEC '05)*, pages 79–86, 2005.

[39] S. Hallsteinsen, M. Hinchey, S. Park, and K. Schmid. Dynamic Software Product Lines. *Computer*, 41(4):93–95, 2008.

[40] M. B. Hmida, R. F. Tomaz, and V. Monfort. Applying AOP Concepts to Increase Web Services Flexibility. *Journal of Digital Information Management*, 4(1):37–43, 2006.

[41] J. Kabanov. Reloading Java Classes 401: HotSwap and JRebel - Behind the Scenes, 2010. Online available at: http://www.zeroturnaround.com/blog/reloading_java_classes_401_hotswap_jrebel.

[42] K. C. Kang, S. G. Cohen, J. A. Hess, W. E. Novak, and A. S. Peterson. Feature-Oriented Domain Analysis (FODA) Feasibility Study. Technical Report CMU/SEI-90-TR-21 ESD-90-TR-222, Software Engineering Institute Carnegie Mellon University, 1990.

[43] C. Kästner, S. Apel, and M. Kuhlemann. Granularity in Software Product Lines. In *Proceedings of the 30th International Conference on Software Engineering (ICSE '08)*, pages 311–320, 2008.

[44] A. Keller and R. Badonnel. Automating the Provisioning of Application Services with the BPEL4WS Workflow Language. In *Proceedings of the 15th IEEE/IFIP Distributed Systems: Operation and Management (DSOM '04)*, pages 15–27, 2004.

[45] K.-K. Lau and Z. Wang. Software Component Models. *IEEE Transactions on Software Engineering*, 33(10):709–724, October 2007.

[46] Y. Li, X. Zhang, Y. Yin, and J. Wu. QoS-Driven Dynamic Reconfiguration of the SOA-Based Software. In *Proceedings of the 2010 International Conference on Service Sciences (ICSS '10)*, pages 99–104, 2010.

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

[48] F. M. Medeiros, E. S. de Almeida, and S. R. L. Meira. Towards an Approach for Service-Oriented Product Line Architectures. In *Proceedings of the 3rd Workshop on Service-Oriented Architectures and Software Product Lines: Enhancing Variation (SOAPL '09)*, 2009.

[49] R. Mietzner and F. Leymann. Towards Provisioning the Cloud: On the Usage of Multi-Granularity Flows and Services to Realize a Unified Provisioning Infrastructure for SaaS Applications. In *Proceedings of the 2008 IEEE Congress on Services (SERVICES '08)*, pages 3–10, 2008.

[50] R. Mietzner, A. Metzger, F. Leymann, and K. Pohl. Variability Modeling to Support Customization and Deployment of Multi-Tenant-Aware Software as a Service Applications. In *Proceedings of the 2009 Workshop on Engineering Service Oriented Systems (ICSE '09)*, pages 18–25, 2009.

[51] V. Monfort and S. Hammoudi. Towards Adaptable SOA: Model Driven Development, Context and Aspect. In *Proceedings of the 7th International Joint*

Conference on Service-Oriented Computing (ICSOC-ServiceWave '09), pages 175–189, 2009.

[52] M. Moon, M. Hong, and K. Yeom. Two-Level Variability Analysis for Business Process with Reusability and Extensibility. In *Proceedings of the 32nd Annual IEEE International Computer Software and Application Conference (COMPSAC '08)*, pages 263–270, 2008.

[53] D. Muthig and T. Patzke. Generic Implementation of Product Line Components. In *Proceedings of the 2002 Net.ObjectDays (NODE '02)*, pages 313–329, 2002.

[54] N. C. Narendra, K. Ponnalagu, B. Srivastava, and G. S. Banavar. Variation-Oriented Engineering (VOE): Enhancing Reusability of SOA-Based Solutions. In *Proceedings of the 5th IEEE International Conference on Services Computing (SCC '08)*, pages 257–264, 2008.

[55] Object Management Group, Inc. (OMG). Deployment and configuration of component-based distributed applications specification (DEPL), 2006. Online available at: http://www.omg.org/spec/DEPL/.

[56] Object Management Group, Inc. (OMG). Quality of Service for CORBA Components Specification, version 1.1, 2008. Online available at: http://www.omg.org/technology/documents/corba_spec_catalog.htm.

[57] Object Management Group, Inc. (OMG). Common Object Request Broker Architecture (CORBA/IIOP), 2011. Online available at: http://www.omg.org/spec/CORBA/3.1.1/.

[58] Object Management Group, Inc. (OMG). Unified Modeling Language (UML) Resource Page, 2011. Online available at: http://www.uml.org/.

[59] Open Service Oriented Architecture Collaboration (OSOA). Service Component Architecture Specifications, 2009. Online available at: http://www.osoa.org/display/Main/Service+Component+Architecture+Specifications.

[60] Oracle. Java Remote Method Invocation (RMI), 2010. Online available at: http://download.oracle.com/javase/7/docs/platform/rmi/spec/rmiTOC.html.

[61] Oracle. Java SE HotSpot at a glance, 2011. Online available at: http://www.oracle.com/technetwork/java/javase/tech/index-jsp-136373.html.

[62] Organization for the Advancement of Structured Information Standards (OASIS). Web Services Dynamic Discovery (WS-Discovery) Version 1.1, 2009. Online available at: http://docs.oasis-open.org/ws-dd/discovery/1.1/os/wsdd-discovery-1.1-spec-os.pdf.

[63] Organization for the Advancement of Structured Information Standards (OASIS). OASIS Web Service Business Process Execution Language (WS-BPEL), 2011. Online available at: http://www.oasis-open.org/committees/tc_home.php?wg_abbrev=wsbpel.

[64] J. Park, J. Kim, S. Yun, M. Moon, and K. Yeom. An Approach to Developing Reusable Domain Services for Service-Oriented Applications. In *Proceedings of*

the 25th ACM Symposium on Applied Computing (SAC '10), pages 2252–2256, 2010.

[65] C. Parra, X. Blanc, and L. Duchien. Context Awareness for Dynamic Service-Oriented Product Lines. In *Proceedings of the 13th International Software Product Line Conference (SPLC '09)*, pages 131–140, 2009.

[66] C. Parra, R. Leano, X. Blanc, L. Duchien, N. Pessemier, C. Taconet, and Z. Kazi-Aoul. Dynamic Software Product Lines for Context-Aware Web Services. In Q. Z. Sheng, J. Yu, and S. Dustdar, editors, *Enabling Context-Aware Web Services: Methods, Architectures, and Technologies*. Chapman and Hall/CRC, 2009.

[67] K. Pohl, G. Böckle, and F. van der Linden. *Software Product Line Engineering: Foundations, Principles, and Techniques*. Springer, Berlin / Heidelberg, 2005.

[68] I. Schaefer. Variability Modelling for Model-Driven Development of Software Product Lines. In *Proceedings of the 4th International Workshop on Variability Modelling of Software-intensive Systems (VAMOS '10)*, 2010.

[69] T. Scheibler, R. Mietzner, and F. Leymann. EAI as a Service - Combining the Power of Executable EAI Patterns and SaaS. In *Proceedings of the 12th International IEEE Enterprise Distributed Object Computing Conference (EDOC '08)*, pages 107–116, 2008.

[70] K. Schmid. A Quantitative Model of the Value of Architecture in Product Line Adoption. In *Proceedings of the 5th International Workshop on Product Family Engineering (PFE '03)*, pages 32–43, 2003.

[71] K. Schmid and H. Eichelberger. EASy-Producer – A Product Line Production Environment. In *Proceedings of the 12th International Software Product Line Conference (SPLC '08)*, pages 357–357, 2008.

[72] K. Schmid and H. Eichelberger. Model-Based Implementation of Meta-Variability Constructs: A Case Study using Aspects. In *Proceedings of the 2nd International Workshop on Variability Modelling of Software-intensive Systems (VAMOS '08)*, pages 63–71, 2008.

[73] K. Schmid and I. John. A Customizable Approach To Full-Life Cycle Variability Management. *Science of Computer Programming*, 53(3):259–284, 2004.

[74] P.-Y. Schobbens, P. Heymans, and J.-C. Trigaux. Feature Diagrams: A Survey and a Formal Semantics. In *Proceedings of the 14th IEEE Requirements Engineering Conference (RE '06)*, pages 139–148, 2006.

[75] N. Siegmund, M. Pukall, M. Soffner, V. Köppen, and G. Saake. Using Software Product Lines for Runtime Interoperability. In *Proceedings of the Workshop on Reflection, AOP and Meta-Data for Software Evolution (RAM-SE '09)*, pages 1–7, 2009.

[76] B. Steffen and T. Margaria. METAFrame in Practice: Design of Intelligent Network Services. In Ernst-Rüdiger Olderog and Bernhard Steffen, editors, *Correct System Design*, volume 1710 of *Lecture Notes in Computer Science*, pages 390–415. Springer Berlin / Heidelberg, 1999. 10.1007/3-540-48092-7_17.

[77] B. Steffen, T. Margaria, V. Braun, and N. Kalt. Hierarchical Service Definition. *Annual Review of Communication, International Engineering Consortium (IEC)*, pages 847–856, 1997.

[78] M. Svahnberg, J. van Gurp, and J. Bosch. A Taxonomy of Variability Realization Techniques. *Software – Practice and Experience*, 35(8):705–754, 2005.

[79] The OSGi Alliance. OSGi - The Dynamic Module System for Java, 2011. online available at www.osgi.org.

[80] The OSGi Alliance. OSGi Service Platform Core Specification, 2011. Online available at: http://www.osgi.org/download/r4v43/r4.core.pdf.

[81] R. F. Tomaz, M. B. Hmida, and V. Monfort. Concrete Solutions for Web Services Adaptability Using Policies and Aspects. *International Journal of Cooperative Information Systems*, 15(3):415–438, 2006.

[82] A. van der Hoek. Design-Time Product Line Architectures for Any-Time Variability. *Science of Computer Programming, Special Issue on Software Variability Management*, 53(30):285–304, 2004.

[83] S. Walraven, B. Lagaisse, E. Truyen, and W. Joosen. Aspect-Based Variability Model for Cross-Organizational Features in Service Networks. In *Proceedings of the 1st International Workshop on Composition: Objects, Aspects, Components, Services and Product Lines (Composition & Variability '10)*, pages 57–63, 2010.

[84] S. Walraven and P. Verbaeten. AO Middleware Supporting Variability and Dynamic Customization of Security Extensions in the ORB Layer. In *Proceedings of the 9th ACM/IFIP/USENIX International Conference on Middleware (Middleware '08)*, pages 121–123, 2008.

[85] World Wide Web Consortium (W3C). Web Services Description Language (WSDL) 1.1, 2001. Online available at: http://www.w3.org/TR/2001/NOTE-wsdl-20010315.

[86] World Wide Web Consortium (W3C). SOAP Version 1.2 Part 1: Messaging Framework (Second Edition), 2007. Online available at: http://www.w3.org/TR/2007/REC-soap12-part1-20070427/.

[87] W. Zhao, B. R. Bryant, C. C. Burt, R. R. Raje, A. M. Olson, and M. Auguston. Automated Glue/Wrapper Code Generation in Integration of Distributed and Heterogeneous Software Components. In *Proceedings of the 8th IEEE International Enterprise Distributed Object Computing Conference (EDOC '04)*, pages 275–285, 2004.