# Service Grid Variability Realization

Jilles van Gurp, Juha Savolainen
*Software and Application Technologies Laboratory*
*Nokia Research Center*
*P.O. Box 407, FI-00045 NOKIA GROUP, Finland*
*{jilles.vangurp|juha.e.savolainen}@nokia.com*

## Abstract

*Variability management has long been recognized as a key part of software product family development. This article builds on this notion by presenting a set of web service related technologies in the context of variability management. Additionally we adapt an existing process for planning variability for use with our technologies. We expect that web service technology, already very successful in the domain of enterprise applications, will emerge as the integration technology of choice for constructing so called product family populations, i.e. populations of products constructed from multiple, independently developed product families.*

## 1. Introduction

This article presents techniques and a process for using them that together may be used to implement variant features in web service grid architectures.

An important reason for using web services and service grids is to abstract away from implementation details such as data persistence, implementation language, etc. so that external applications may use them with the bare minimum of assumptions about implementation and internal behavior of the web services in the architecture. Web services are now commonly used in enterprise software systems to, for example, integrate software from different vendors, or to integrate legacy systems in new applications.

Web service technology is also likely to be used for integrating product family products to create a population of product families [19]. Previous research has already focused on planning variability in software product families [4, 6, 10, 17]. This article extends this research so that it may be used in combination with web service technology to create populations of product families with a specified level of variability using web service technology.

### 1.1 Service Grids

Service grids combine web services and grid computing. In grid computing, software is run on a distributed computer consisting of multiple, general purpose computers connected by a network. The distributed computer is referred to as a grid and the individual computers in the network are referred to as nodes.

Grid computers have some nice characteristics that make them interesting for application development:

- They are low in cost because they can be created from ordinary server hardware (or even existing desktop machines).
- They provide fault tolerance. If one node fails, the other nodes can compensate for the loss.
- They provide scalability. The capacity of the grid can be increased simply by adding nodes. This only works for software that can be implemented in a distributed fashion, though.
- They provide flexibility because grids can change and adapt dynamically to changing circumstances (e.g. adding new nodes, moving software services between nodes, etc).

Strictly speaking, the techniques discussed in this article are not specific to service grids. However, the assumption that services are deployed across many machines makes the techniques more useful because the address of the service is run-time variable as the service grid dynamically changes the service grid configuration to match demand for particular services. Unlike clustering where identical servers share the load and a load balancing router acts as a facade, the nodes in a grid are not identical. The grid management software starts and stops web services on nodes as required by the run-time context.

### 1.2 Web Services

Web services, on the other hand, have been adopted on a large scale since the introduction of SOAP and XML in the late nineties. A Web service is a software system designed to support interoperable machine-to-machine interaction over a network. It has an interface described

in a machine-processable format, usually WSDL (Web Service Description Language). Other systems interact with the Web service in a manner prescribed by its description using SOAP messages, typically conveyed using HTTP with an XML serialization in conjunction with other Web service related standards [3]. Within the scope of this article, we adopt this narrow definition of a web service provided by the World Wide Web Consortium (W3C). There are also non W3C means of implementing web services. For example, XMLRPC and REST [7] based protocols are used in industry as well.

Web services are stateless. Between SOAP message exchanges, no state is maintained on the server-side. The grid node receives the soap request, processes it and sends a message back. While the processing may include storing data in a database, the server itself is stateless. Being stateless makes it easy to provide scalability (e.g. by adding more nodes to the grid).

### 1.3 Problem Statement

The identification of variant features during requirements gathering and the subsequent realization of them in the realization phase is the backbone of software product family development [20][4]. Software product family development has been characterized as the most successful reuse strategy so far.

Van Ommering makes a distinction between product families and product populations [19]. A product population consists of multiple, independently developed product families that need to be integrated. Unlike product lines where products share the same architecture in which variability can be planned, product populations do not have an overall architecture. To tackle the problem of integrating components from different product families, Van Ommering introduces an architecture language, KOALA that can be used to specify configurations of components. A KOALA configuration consist of provided & required component interfaces and so called switches that are used to translate the internal provided variability to the KOALA level where it may be exploited in product configurations. Essentially, KOALA is an integration platform that provides a set of variability realization techniques that may be used to construct product configurations. KOALA is specific for the Philips context of embedded, C based consumer electronics software.

In recent years, web services have emerged as a means to integrate independently developed software components (often from different vendors). The problem tackled by web services is remarkably similar to the problem tackled by Van Ommering's KOALA. In order to construct software products from web service technology, multiple, independently developed software components are web service enabled and hooked up in a service grid (which may cross organizational boundaries). Service grids are distributed software systems consisting of independently developed software systems whose only commonality is that they can be accessed as a web service.

Since web services and service grid technology are still rather new technologies, relatively little research has been done into how to develop software with these technologies. We argue that designing service grid applications is very similar to designing software products in a software population, as described by van Ommering. The service grid software architect has to combine multiple software products, each enabled with web service technology and capable of being executed in a service grid context. Effectively, the various services in a service grid form a product population from which new products may be created by simply combining them. Building software applications in a service grid involves selecting, adapting and reusing existing services; adding new services and implementing glue code and client code.

Despite the similarities, there are also some differences. The ADL and technical solutions van Ommering proposes are appropriate for the domain of embedded system software, not for web services in a service grid.

So, there is a need for new methodology and techniques to address intra organization reuse in a web service context. Web service technology is the platform of choice for this type of reuse and there is a need for methodology and technology to adjust software engineering practice.

We claim that existing variability management techniques that have already been used successfully in the context of product families and product populations are well suited to fill this need. Service grids have their own specific ways of implementing variability.

In this article, we explore a set of variability realization techniques specific for service grids. We present a list of readily applicable service grid specific technologies that may be use to realize variant features in service grid applications. Additionally we present a process for using them based on our earlier work [17]. In this earlier article, we presented a taxonomy of variability realization techniques from which software product family architects may select when designing software product family architects with specified level of variability. This article specializes that approach for use with service grids.

### 1.4 Remainder of This Paper

Section 2 presents concepts and terminology our categorization used for presenting the variability realization techniques. Section 3 presents nine techniques; section 4 outlines a process for using them.

Section 5 presents related work and we conclude the article in section 6.

## 2. Variant Features in Service Grids

Variability management is considered to be of great importance to maximize the reuse in software product families [1][4]. The goal of the product family approach is to simplify product development by providing a flexible, reusable product family core. Product development is then a matter of exploiting the provided variability in the product family core (selecting from existing components, implementing new components, etc).

### 2.1 Terminology

In an earlier publication [17], we introduced terminology and a taxonomy for classifying variability realization techniques: techniques that may be used for the technical realization of variant features. In this paper we apply the terminology and, to some extent, the taxonomy. Our earlier work positions variability management as a means to identify, plan, design and realize variant features in software product families. However, the terminology mostly applies to other types of software as well.

During the requirements phase it is identified that a particular feature needs to be variant (e.g. the feature is optional or alternative variants of the same feature are identified). Then the variant feature is constrained. This means that the requirements engineer determines when the variant feature should be bound to a specific variant. Binding time refers not to the development phases but to very specific transitions any software system goes through:

**Architecture Derivation.** In software product family development, new software products may be based on an existing software architecture. This software architecture can support variability at the design level and can present the designer with design decision such as which components to use, redesign or omit in the product design. In a service grid application architecture, architecture derivation consists of selecting service implementations and WSDL descriptions. Some services may be reused; some may require product specific implementations. Additionally, there are various standardized service components that may be incorporated into the application. Essentially, the set of web service specifications provided by, for example, the W3C and OASIS provide a high-level web service product line architecture from which web service based product architectures may be derived. Toolkits and libraries such as the Globus toolkit, Microsoft's WSRF.Net or Apache Muse that implement these specifications may be used as reusable assets. The selection of which components of what toolkit to use is of course a very important architecture decision.

**Compilation.** During compilation source code written by the software developers, is translated into executable code. The build process generally is configurable. In the context of a web service, compilation includes generating stub code from a WSDL file. Any variability technique requiring changes to the WSDL is thus bound during compilation.

**Linking.** Linking is a process where it is decided which software libraries/classes/objects are used. Linking can be static (as part of the compilation process) or dynamic (which means that linking takes place during application startup or during run-time). For a web service, linking is similar to looking up the web service endpoint for a particular service using e.g. an UDDI (Universal Description, Discovery and Integration) registry.

**Software startup.** When an executable software system is started, parameters, configuration files, etc. are read. The purpose of a service grid is, amongst others, reliability. Restarting the grid is, or should be, a rare event. However, web services may be installed into the service grid dynamically and have their own lifecycle (start & stop of the service).

**Run-time**. Finally while a software system is running, parameters may set, plugins may be loaded, etc. For a web service this would mean calling it (by sending a SOAP message) with specific parameter values that are defined in the WSDL description of the service.

Deciding on a binding time is said to constrain the variant feature.

A second way to constrain a variant feature is to associate stakeholders with the various stages a variant feature goes through:

- **Identification & introduction**. It is identified that there is a need for variation; a variability realization technique is selected and variation points (i.e. a concrete representation (or representations) of the variant feature in the development artifacts.) are added to the software.
- **Population**. Variants of the variation points are added to the software.
- **Binding**. Specific variants are selected and bound to the variation points.

In the context of service grid applications, three different types of stakeholders are involved:

- **Service provider**. This stakeholder provides a specific service and is responsible for its design and implementation.
- **Service consumer**. This stakeholder uses the services offered by the service provider.

- **Service mediator**. Service mediation is the process of connecting the provided service to its consumer. In a service grid, service mediation may involve looking up the service endpoint, reconfiguring the grid to adjust dynamically for changing capacity needs, delivering asynchronous messages to subscribers, etc.

## 2.2 Categorization of Techniques

This article presents variability realization techniques for use in service grid applications. As we argued in [17], managing variability involves identifying & constraining the variant features and then selecting a suitable technique. To facilitate this selection we present the techniques in this paper using the following categorization:

- Intent. A brief description of the purpose of the technique.
- Motivation. Discusses why the technique should be used.
- Solution. A brief overview of how the technique works.
- Constraints. Answers the constraint questions listed above.
- Consequences. Discusses positive and negative impact of using the technique.
- Example. An example of applications of the technology.

This categorization is similar to the categorization we used in [17]. In addition, it is very similar to but not the same as the formats used in the design pattern community (e.g. [5] and [8]. We believe this format is best for the problem at hand: selecting a technique based on analysis of variant features.

# 3. Variability Realization Techniques

In this section, we present an overview of variability realization techniques that are specific for service grids.

## 3.1 Service Lookup

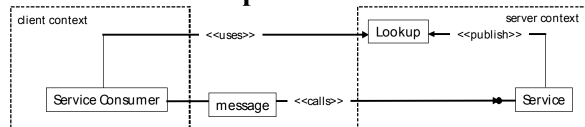

Fig 1. Consumer looks up the service endpoint and sends message.

**Intent**. Allow a variable implementation of a particular service to be used by a particular application.

Motivation. Service lookup makes it possible to implement the consumed web service separately from the calling application. An additional advantage is that the web service implementation may be replaced by a new implementation without changing the calling implementation.

**Solution**. Select the web service implementation by looking up the endpoint for the service at run-time using a lookup service. Service lookup is a crucial component of the broker pattern that is discussed in [5]. The broker pattern underlies technologies such as CORBA, DCOM, RMI, etc.

**Constraints**. New variants in the form of alternative service implementations for a particular service interface are the responsibility of the service providers. However, the variation point these variants bind to is owned by the service consumer. The binding happens at run-time when the mediator looks up the service provider endpoint on behalf of the service consumer.

**Consequences**. A service interface (usually a WSDL interface) must be agreed upon before creating the client application that will consume the service. This interface will need to be registered in a lookup service which complicates deployment (service must be registered; clients must be configured with the lookup service endpoint), system architecture (the lookup service is an extra component) and usage of the service because the client must contain extra functionality to do the service lookup of the service.

**Example**. UDDI (Universal Description, Discovery and Integration) registries are commonly used to provide this type of variability. Service providers register new services at the UDDI registry. Service consumers look up the services in the UDDI by name or by interface. Much of the complexity of registering services in UDDI and using UDDI registries is typically handled by application servers and generated code from integrated development kits such as Microsoft Visual Studio.

## 3.2 Client-side Proxy

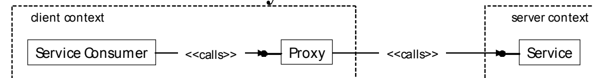

Fig. 2 The service consumer calls a client-side proxy, which makes the call to the service on its behalf.

**Intent**. Allow service lookup without exposing it to the client code in order to make (dynamic) changes in the service operation transparent to the client.

**Motivation**. Having service consumers explicitly take part in the lookup functionality, partially ties the choice of which variant implementation is chosen to the service consumer implementation. Additionally, as noted previously, some complexity is introduced when the service consumer has to perform a service lookup before using the service. In cases where there is a one-to-many relation between the service on one hand, and the service consumer on the other hand, changing the service consumer implementation, or even configuration is undesirable.

**Solution**. Use a proxy library to insulate service consumers from the service lookup and binding. Embed the service lookup functionality in the library.

**Constraints**. Essentially this technique is very similar to the lookup technique with one important difference:

service lookup becomes implicit. The client is presented with a client proxy by the service provider that performs the lookup on behalf of the client as part of using the service rather than that the client initiates a lookup and then calls the server. From the point of view of the service consumer, there is no variation it just calls the web service through the proxy. The variation is embedded in the proxy and adding new variants requires replacing it. The binding of the available variants happens at run-time (though transparent to the service consumer).

**Consequences**. As with the lookup service, some functionality is required to do the lookup. The main difference is that the functionality is part of the client stub rather than the client itself. An additional disadvantage is that the client has no control over the lookup. In general, there may be multiple, suitable service implementations conforming to the same interface with different behavior that may be looked up with an appropriate query. Using this technique, however, hides the lookup from the client and makes it impossible influence the lookup.

**Example**. An example of where this technique is beneficial is mobile devices. Such devices have limited, unreliable connectivity depending on the quality of the connection different lookup strategies may apply. Rather than embedding this logic in client applications, it may be better to implement a smart proxy. For example, an application that sends information to a remote service may still be usable when the device is out of range of a base station if an intermediate proxy stores the information locally until it can be sent when the device is connected again.

## 3.3 Façade / Gateway

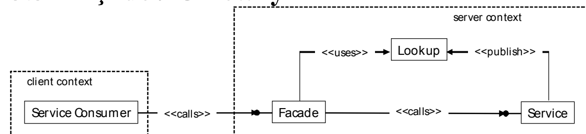


Fig. 3 The service consumer calls a façade, which looks up the correct endpoint and passes the call to it.

**Intent**. Allow single point of entry to the service grid to hide that the grid has multiple instances of the same web service running on multiple nodes.

**Motivation**. A service grid typically hosts a large number of web services that can be mapped to service endpoints in the grid dynamically. The service consumers must somehow connect to a specific service on a specific service endpoint in the grid. Normally a lookup service can facilitate this. However, as discussed in the lookup technique, this complicates using the service. Additionally, SOAP enabling clients is not always feasible or desirable. E.g. in resource limited devices such as mobile phones more efficient simple binary protocols may be preferred to the relatively bandwidth and processing intensive SOAP protocol. So using a client side proxy is not an option.

**Solution**. Use a gateway that provides load balancing and routing capabilities for the service grid clients. The gateway provides a single point of entry for clients. This gateway may itself be a SOAP service but it may also provide other means of access. For example, it might implement the XMLRPC (http://www.xmlrpc.com/) protocol or a REST based protocol [7].

**Constraints**. This technique is similar to the client side proxy. The main difference is that the façade is server side instead of client side.

**Consequences**. This creates a single point through which all requests pass. This single point may become a bottleneck. Additionally, as noted in [7], introducing new layers in a network architecture generally introduces additional latency because the data has to pass through an extra layer.

**Example**. Google offers several web services to its customers. For example, their ad words functionality is exposed as a web service at https://adwords.google.com/api/adwords/v3. This convenient address is easy to remember and unlikely to change. However, it is unlikely to be the actual endpoint for the web service. Google is known to use a very large cluster of computers to host its various services. The computer at the endpoint cited here is likely to be just a gateway. Even though the Google cluster is probably not based on service grid technology, as discussed in this article, it is similar enough to serve as an example.

## 3.4 Configuration Parameter

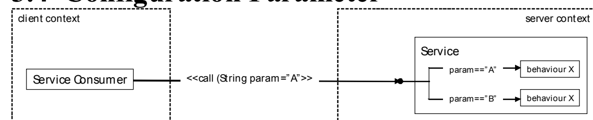


Fig. 4 A parameter in the service call determines which behavior is executed.

**Intent**. Provide information to the web service to select the right variant

**Motivation**. Some variation does not require elaborate mechanisms that introduce configuration & management overhead. For this type of variation, client code may simply specify what variant to use by passing it that information in the form of a simple parameter.

**Solution**. Add a parameter to the operations defined in the WSDL interface to pass the configuration information.

**Constraints**. The variation point is provided by the service provider. The logic for interpreting the value and binding to the right value is executed at run-time. Depending on how this logic works it may be

necessary to modify this functionality to support new variants.

**Consequences**. The semantics of the interface are obscured by the additional parameter which exposes implementation details (i.e. related to the selection of variants).

**Example**. The Google web service, which may be used by application developers to include Google search in their applications, includes a boolean parameter in the API to enable safe search, i.e. the filtering of inappropriate search results.

### 3.5 Turn Parameters into WSRF Resources

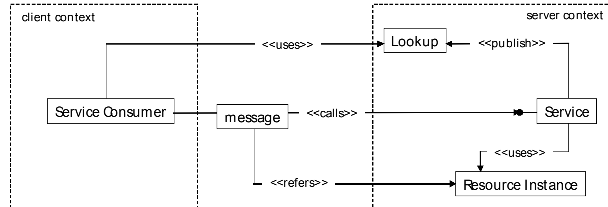


Fig. 5 The message refers to a resource endpoint instead of including a parameter with a id.

**Intent**. Convert the entity identified by a parameter that is present in several web service interfaces to separate web service.

**Motivation**. The parameter is used to refer to an entity known to both the web service and the web service consumer. The presence of the same parameter in multiple web services indicates that the entity is a shared resource. Making that entity available through a standard interface makes this more explicit and makes the resource available for manipulation by other web services as well. Additionally, having a standard interface prevents service implementations from including implementation specific technology for accessing the entity denoted by the parameter.

**Solution**. Convert the entity denoted by the parameter into a WSRF resource. The WSRF (Web Service Resource Framework) provides a standardized interface for manipulating (lifecycle management, reading, and setting properties) resources such as database objects, devices or even the service grid itself. The web service accesses the WSRF resource through its web service interface. This keeps the service implementation free from implementation specifics of the resource implementation. Consequently, new resource types may be added without requiring any change to the services that use them.

**Constraints**. The variation points in this case are references to resource endpoints in the services that use them. These references consist of functionality to send and receive message to and from resource endpoints (i.e. the variants). So the variation points are introduced by the service producer. Services may create new resource instances at run time (through the WSRF interface) so the service provider is also responsible for populating the variation point. Finally binding a service to a particular resource instance is done by embedding a reference to the resource endpoint in the service call. This is done at run-time by the service consumer.

**Consequences**. Client code will have to look up the right WSRF resource. In practice, this means that a WSRF enabled web service toolkit is used for generating client stubs, for example Apache WSRF (http://ws.apache.org/wsrf/) or the Globus toolkit (http://www.globus.org). Then toolkit specific glue code provides the interaction with the resource. Using a web service interface instead of embedding the functionality in the web service also involves some overhead. A call to a web service interface is many times slower than, for example, calling a Java method.

**Example**. A simple example of this technique is the common use case that a web service is used to manipulate a business object that persists to a database. The business object is identified by a primary key which the web service consumer provides to the web service using one of the web service parameters. Likely, the logic implemented by the web service involves reading & writing the properties of the business objects (e.g. using SQL statements). Other related web services may implement similar mechanisms so it is beneficial to make this functionality available in a separate web service. Instead of inventing a custom interface, the WSRF interface should be used for this. At a later stage, new resource implementations in the form of an alternative database backend (e.g. different table layout) may be added. Also by encapsulating legacy components as a WSRF resource, these components may be integrated into new applications.

### 3.6 Service Grid Level AOP

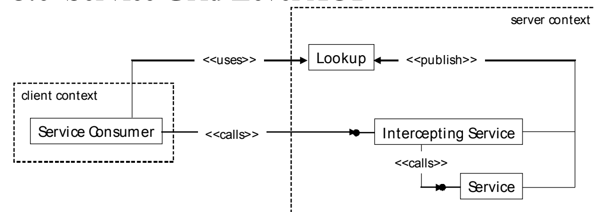


Fig. 6 The lookup for the service results to an endpoint to an intercepting service which makes the call to the actual service. All of this is transparent to the service consumer.

**Intent**. Insert new functionality in existing web service configurations without modifying service consumers or service providers.

**Motivation**. As noted earlier, modifying client code is not always feasible or desirable, especially in the case of many deployed service consumers that are hard to upgrade. Additionally, modifying existing service providers to implement new functionality may not be desirable either for several reasons:

- There may be service consumers that need the existing behavior.
- The service is provided by an external entity.
- Not all service consumers need the new functionality.

Yet, there may be a need to modify the functionality of the web service.

**Solution**. Redirect the web service calls through a third proxy web service that processes the message, calls the target end point on behalf of the service consumer, intercepts and processes the response and sends the processed response back to the caller. This mechanism is very similar to Aspect Oriented Programming [12] where programmers 'intercept' program execution to insert extra functionality.

**Constraints**. This technique has an implicit variation point: the connection between service provider and consumer. These connections are established, at run-time, by the service mediator. Variation is achieved by rerouting messages through a third service, also at run-time.

**Consequences**. The level of indirection may have a performance penalty since the number of web service calls increases for each proxy added.

**Example**. This type of message interception is supported by many SOAP stacks. For example, Apache Axis has the notion of handlers. A SOAP message may pass through many handlers before arriving at its destination. But even without relying on this kind of support, it is possible to just set up an intermediate web service. A simple example could be verifying if the client is properly authenticated before passing the SOAP message on to a service that should only be used by properly authenticated clients but does not have any functionality for this it self.

## 3.7 Service Adaptation or Mediation

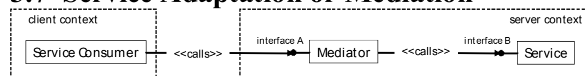


Fig. 7 The service consumer calls a mediating service implementing interface A. This service makes a call to another service using interface B that the service consumer is incompatible with.

**Intent**. Allow an incompatible service consumer and web service to work together.

**Motivation**. The situation that an independently developed software component provides the required functionality but is not compatible (e.g. because the interface does not match) is quite common. Especially in a service grid, where multiple, independently developed web services need to be combined into service grid application this scenario, this is quite likely.

**Solution**. Introduce a mediating web service to work around the incompatibility. The mediating web service implements an interface that the client understands and it implements this interface by using the incompatible web service in the appropriate way. It encapsulates the mediation logic required.

The problem that service mediation solves is not unique to web services. In Koala, the architecture description language Van Ommering describes [19], components are connected through 'glue modules' intended to mediate interface incompatibilities between components. Service mediation is an important concept in any web service architecture

**Constraints**. Service mediation adapts the provided variability (by the service provider) to the required variability (by the service provider). A mediating service provides an alternative interface to an existing service. The variation point is the original service interface. The variants, consisting of mediator service interfaces are added by the mediator. Binding of a service consumer to the service provider through the mediating service happens at run-time.

**Consequences**. Service mediation may not always be possible (or desirable). The resulting functionality is similar to glue code or script code mentioned in [13] and is likely to be specific for the service consumer - provider combination.

**Example**. Synapse (http://incubator.apache.org/synapse/), an open source framework for service mediation, provides functionality for implementing service mediation services. The framework facilitates such things as message transformation; message routing and even load balancing.

## 3.8 Role-oriented Web Services

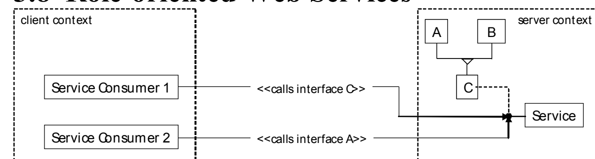


Fig. 8 Two service consumers use the same service but through a different (subset of) the interface.

**Intent**. Make it possible for existing software to interact with a new web service.

**Motivation**. The API (application programming interface) of a web service component allows the service to be used in various interactions with other web services or service consumers. Some of these interactions follow a pattern that is not specific for the web service implementation. Using a standardized interface for these interactions enables any clients that understand these interfaces to use the web service.

**Solution**. The WSDL (Web Service Description Language) describes the full interface of the web service.

The WSDL import functionality allows developers to import WSDL fragments. Alternatively, the relevant portions of WSDL can be pasted into the web service description. Either method allows the reuse of existing

specifications for parts of the web service interface. Reusing existing interfaces allows existing web service clients that are already compatible with these interfaces to interact with the new service as well. This mechanism is similar to role based programming popularized in e.g. [14] and commonly used in e.g. Java and .Net based computer systems.

**Constraints**. The variation points in this technique are references to services that implement the role interface. The decision to include such references is the responsibility of the service consumer. The variants consist of any service that implements an interface that includes the role interface. New variants may be added at run-time by the service provider. Binding, i.e. selecting an interface implementation is done at run time, e.g. through a lookup service.

**Consequences**. The current version of the WSDL specification has limited support for importing external WSDL interfaces. The upcoming 2.0 version will improve this but meanwhile a certain amount of copy paste reuse cannot be avoided, when applying this technique.

**Example**. Many of the web service specifications standardized by the World Wide Web Consortium (W3C) or OASIS standardize common interfaces. For example WSRF (discussed earlier) standardizes interfaces for the manipulation of resources; WS-Notification standardizes interfaces for publish subscribe type interactions. The benefit of conforming to such standardized interfaces is interoperability with other, independently developed software that conforms to these specifications as well. Currently available tools for creating web service based applications all include some level of support for the before mentioned standardized interfaces. Consequently, web services implemented using these tools are likely to expose multiple standardized interfaces and be accessible in a standardized way.

### 3.9 Alternative Service Interfaces

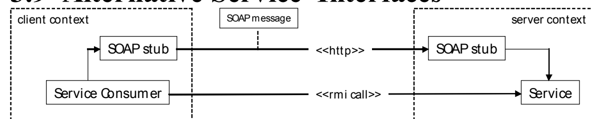


Fig. 9 The service consumer has two options for using the service: a SOAP interface and a second interface (e.g. Java RMI based). If the service consumer is implemented in Java and running in the same local network as the service, it may use the more lightweight RMI instead of the SOAP interface.

**Intent**. Improve performance by avoiding the use of heavy weight protocols such as SOAP.

**Motivation**. SOAP is an integration technology. Using SOAP makes sense when calling other applications on different network nodes, implemented using different technologies. It is known to introduce significant overhead, however. Therefore, web services interacting with each other might benefit from alternative means of interaction.

**Solution**. These alternatives means of interaction may range from XMLRPC to more traditional remote procedure forms such as RMI, CORBA, etc. The use of the SOAP service interface then becomes optional, required only if the other techniques are not supported.

**Constraints**. The service provider needs to provide an alternative ways to access the functionality. The variation point in this case is the native interface of the functionality (e.g. a Java interface or a C header file); the variants are the various means of accessing this interface (e.g. a SOAP server stub, an XMLRPC handler, etc).

**Consequences**. The client and server functionality needs to be extended with functionality that decides when to use what form of remote procedure calls.

**Example**. Recently, the reference implementation of the Java Business Integration (JBI) standard was finalized (http://java.sun.com/integration/). The aim of this technology is to separate the binding of web services from specific web service protocols. Web service implementations use a lookup mechanism (part of the JBI infrastructure) to find each other. The JBI middleware then takes care of delivering messages from one service to another using an appropriate protocol (e.g. SOAP, Java Remote Method Invocation (RMI), CORBA, etc.).

## 4. Selecting the Right Technique

The list of techniques in the previous section may all be used to realize required variability in a service grid application. In this article, we suggest that one of the things to consider in this decision is the variability constraints associated with the variant feature under consideration. The process we propose for this (based on our previous work [17]) is:

- Identify the variant features relevant at the web service architectural level. There are several ways to do this. There appears to be a lot of consensus that domain analysis and feature diagrams in particular are suitable for identifying and documenting variability. FODA [10], for example, includes a feature diagram notation, which we specialized informally in [17] Riebisch et al. present a similar UML based notation [15]. All of these notations organize the requirements into a feature hierarchy where the nodes in the hierarchy represent variability (i.e. the variant features).
- Constrain the variant features. Using the terminology from section 2 related to introduction of the variation point(s), population of the variation point with variants and binding of the software to a particular variant.

- Using the list of provided techniques and the constraints, select a technology that satisfies the constraints. The constraints listed with the techniques above may be of use for this purpose. However, additional issues such as whether the added complexity is worth the benefit of exactly fitting the constraints should also be taken into account. Other quality attributes such as performance, maintainability, etc. should also be taken into account. Our list of technologies does not explicitly include a discussion on quality attributes.
- Minimize the number of techniques used unless the constraints require their usage. Minimizing the number of techniques is required to keep the level of complexity under control. If necessary, flexibility should be traded off for uniformity and simplicity.

## 5. Related Work

**Product families & Variability**. Various recent publications, e.g. [20] [9] [6] [4], have established product family research as a separate discipline in the software engineering community. Product families are seen as one of the most successful ways of achieving reuse when developing in the large. Recognizing that modern software development is increasingly about integrating large software components across organizational boundaries, Van Ommering et. al. developed KOALA, an architecture development language designed to create products from product populations [19]. Our article focuses on specific integration technology (web services and service grids) that the authors believe is going to be as important in the software product family community as it already is in the enterprise application arena where all large vendors (Sun, Microsoft, IBM, etc.) have made large investments in web service technology. A key component of product family development is planning for variability (e.g. [4] and [20] claim this).

Additionally, as noted earlier, approaches such as FODA [10] and derived approaches such as FORM [11] and the UML based notation in [15].

In [16], a feature diagram notation is used to identify variability in web service architectures. However, their approach focuses on the user's point of view instead of integrating web services from multiple sources, such as in this article.

**Architectural styles**. As such, the technologies presented in this article are not new and should in fact be very familiar to software architects who most likely employ most of these techniques in practice. Most of these techniques are variations of techniques that are also used in other types of architectures. Nearly all of the techniques are in one form or another derivatives or instances of the patterns and architectural styles popularized in [8] and [5]. However we present them in the context of variability management, an angle that is generally not considered in the pattern community and focus on service oriented architecture which is emerging as the common architectural style for programming in the large. For this reason our way of presenting the techniques is also slightly different.

In his dissertation on network architectural styles [7], Fielding, co founder of the Apache Foundation and author on, amongst others, the HTTP specification, outlines the architectural principles of the WWW that he helped build. He refers to this architectural style as REST (Representative State Transfer). Most of the REST concepts that Fielding apply to service grids as well. For example, Fielding discusses gateway and proxy components, both of which are in our overview of techniques.

**Role oriented programming**. The notion of role oriented programming was popularized first in the object oriented programming community. For example Reenskaug et al. [14], published about their OORAM methodology where classes extend multiple role classes, each representing a specific role in an interaction with other objects. Catalysis [18] is a similar method that works on the same principle. Role oriented programming recognizes that objects can be used in different types of interactions with other objects. Each of these interaction types is associated with a subset of the interface of the object. This principle extends to web services where a number of standardized role interfaces are emerging (e.g. WS-Notification, WS-Security, etc). Our role oriented web services technique is based on this notion.

## 6. Summary

In this article, we have presented list of techniques and accompanying process intended for realizing variability in service oriented architectures. Strictly speaking, most of our techniques are not specific to service grids. However, service grids introduces a lot of variation (e.g. in endpoint location) that makes using our techniques a necessity since e.g. hardwiring endpoint addresses (which is common in stand alone web service implementations) ceases to be an option.

Variability management has proven to be an essential component of most software product family approaches. We expect that this will continue to apply when web service technology is adopted as the primary integration technology for defining product populations of service enabled components distributed in corporate service grids that may even cross organizational boundaries.

Our work contributes to promoting this by relating a set of web service technologies to the variability

terminology that is already used in the context of product family development. Thus, the process such as outlined in section 4, which is already a part of some traditional product family development methods such as described in e.g. [20] and [4], may be used to explicitly design for variability and make informed decisions based on the requirements with respect to technology selection and usage.

**Future work.** The process outlined in section 4 must be extended and embedded in existing product family methodology. Validating such methodologies should be part of such research. Furthermore, our list of techniques is far from complete (nor is it intended to be at this point). Similar to the pattern community more work could be done to investigate, categorize, and describe web service variability realization techniques. Finally, the currently emerging set of web service standards still has a large number of issues with respect to usability, performance, complexity etc. We expect that there will continue to be rapid improvements and innovations in this area for the next few years.

## 7. References


[1] F. Bachmann, L. Bass, “Managing variability in software architectures”. proceedings of the ACM Symposium on Software Reusability: Putting Software Reuse in Context, pp. 126-132, 2001.

[2] M. Baker. "Ian Foster on Recent Changes in the Grid Community", IEEE Distributed Systems Online, 5(2), February 2004.

[3] David Booth, Hugo Haas, Francis McCabe, Eric Newcomer, Michael Champion, Chris Ferris, David Orchard. "Web Services Architecture", Web Services Architecture Working Group, http://www.w3.org/TR/ws-arch/, February 2004.

[4] Jan Bosch, “Design & Use of Software Architectures - Adopting and Evolving a Product Line Approach“, Addison-Wesley, 2000.

[5] F. Buschmann, C. Jäkel, R. Meunier, H. Rohnert, M. Stahl, “Pattern-Oriented Software Architecture - A System of Patterns“, John Wiley & Sons, 1996.

[6] P. Clements, L. Northrop, “Software Product Lines - Practices and Patterns”, Addison-Wesley, 2002.

[7] R. T. Fielding, " Architectural Styles and the Design of Network-based Software Architectures", Ph. D. thesis, University of California, Irvine, 2000.

[8] E. Gamma, R. Helm, R. Johnson, J. Vlissides. "Design Patterns - Elements of Reusable Object Oriented Software". Addison-Wesley, 1995.

[9] M. Jazayeri, A. Ran, F. Van Der Linden, “Software Architecture for Product Families: Principles and Practice”, Addison-Wesley, 2000.

[10] K. C. Kang, S. G. Cohen, J. A. Hess, W.E. Novak, A.S. Peterson, “Feature Oriented Domain Analysis (FODA) Feasibility Study“, Technical report CMU/SEI-90-TR-21, Software Engineering Institute, Carnegie Mellon University, Pittsburgh, PA.

[11] K. Kang, S. Kim, J. Lee, E. Shin, M. Huh. FORM: A Feature-Oriented Reuse Method with Domain-Specific Reference Architectures, "Annals of Software Engineering", 5(5), pp. 143-168, September 1998.

[12] Kiczalez, J. Lamping, A. Mendhekar, C. Maeda, C. Lopes, J-M. Loingtier, J. Irwin, "Aspect Oriented Programming", Proceedings of ECOOP 1997, pp. 220-242, 1997.

[13] J. K. Ousterhout, "Scripting: Higher Level Programming for the 21st Century", In IEEE Computer Magazine 1998 ; 31(3):23-30.

[14] Reenskaug, T., P. Wold and O.A. Lehne "Working with Objects; The OORam Software Engineering Method", Prentice Hall, 1995.

[15] M. Riebisch, K. Böllert, D. Streitferdt, I. Philippow, Extending Feature Diagrams With Uml Multiplicities, proceedings of Integrated Design and Process Technology, IDPT-2002, 2002.

[16] Silva Robak, Bogdan Franczyk, "Modeling Web Services Variability with Feature Diagrams", In Web, Web-Services, and Database Systems, pages 120--128. Akmal B. Chaudrin, Mario Jeckle, Erhard Rahm and Rainer Unland, 2002.

[17] Mikael Svahnberg, Jilles van Gurp, Jan Bosch. "A taxonomy of variability realization techniques", Software Practice & Experience, 35(8), pp. 705-754, 2005.

[18] D. D’Souza, A.C. Wills, Composing Modeling Frameworks in Catalysis. in Building Application Frameworks - Object Oriented Foundations of Framework Design, M. E. Fayad, D. C. Schmidt, R. E. Johnson (eds.), John Wiley & Sons, 1999.

[19] R. van Ommering, Building product populations with software components, proceedings of the 24rd International Conference on Software Engineering, pp. 255 - 265, 2002.

[20] C. T. R. Lai, D. M. Weiss, “Software Product-Line Engineering: A Family Based Software Development Process”, Addison-Wesley, 1999.