Nguyen, T., Colman, A., & Han, J. (2012). Enabling the delivery of customizable web services.


Available from: http://dx.doi.org/10.1109/icws.2012.23.
Enabling the Delivery of Customizable Web Services

Tuan Nguyen, Alan Colman, and Jun Han
Swinburne University of Technology, Melbourne, Australia
{tnnguyen, acolman, jhan}@swin.edu.au

Abstract—Due to differences in consumer requirements, a Web service usually has multiple service variants for use in different business contexts. In such situations, delivering customizable services helps increase efficiency not only in service description and publication but also in service consumption. However, existing approaches for providing customizable services enforce the tight coupling between providers and consumers. Nor do they take into account recursive nature of service customization. Consequently, the approaches hamper the widespread use of customizable services in SOA. In this paper, we propose a language, namely Web Service Variability Description Language (WSVL), which formalizes the customization interface between providers and consumers using the XML technology to address these problems. We also describe a reference architecture for service deployment and a service engineering technique which together support the provisioning of WSVL-based customizable services. A proof-of-concept prototype system is introduced to demonstrate the feasibility of our approach.

Keywords: Customizable services, service variability, service description, service engineering, service provisioning.

I. INTRODUCTION

Web services are fundamental building blocks in developing modern business applications. By enabling interoperability using a stack of XML-based standards, Web service technologies facilitate business integration within and across organizations. Furthermore, the encapsulation of business functionalities in the form of services increases reuse in business application development, thus paving a way to cost reduction, agility and quality improvement.

As an integral part of business applications, services are often coarse-grained and satisfy multiple purposes from different consumers [1, 2]. To improve efficiency in service description, publication and consumption, such services are usually customizable to allow consumers to derive specific service variants matching their application context [3-5]. Providing customizable services is very challenging due to the following three reasons.

Firstly, service customization is a complex process due to a high number of customization options, as well as their inclusive and exclusive dependencies. Such dependencies are usually described using natural language which makes it harder for consumers to perform valid customizations.

Secondly, it is imperative to decrease the extent of coupling between providers and consumers. Service customization is usually performed following an agreed procedure between providers and consumers. For existing approaches, these procedures necessitate the use of a specific technology or an implicit agreement. For instance, in Stollberg’s approach [3], consumers are restricted to using the same service modeling platform (i.e. the Eclipse Modeling Framework) as providers [6]. Or Liang [4] presumes that consumers somehow know the endpoint for invoking customization, as well as how to format and exchange customization messages. Technology specificity and implicit agreement result in tight coupling between providers and consumers. Keeping in mind the many-to-many relationship between consumers and providers, as well as the diversity of service platforms, tight coupling will significantly hinder the use of customizable services.

Thirdly, service customization might be a recursive process. There are cases in which customizing a customizable composite service requires the customization of partner services which again require the customization of other services and so on. In our research, we refer to such dependencies between customizable services as variability inter-dependencies [7]. In highly interdependent service ecosystems, the ability to capture these dependencies is essential to enable recursive service customization.

So far, there are only few works supporting service customization in literature (e.g. [3, 4, 8]). However, these works only partly address the first challenge, while enforcing the tight coupling and leaving the third challenge unconsidered. In previous work [5], we have addressed the first challenge by exploiting the feature modeling technique from the Software Product Line (SPL) [9]. In this paper, we advance the approach to address the remaining challenges.

In particular, to enable loose coupling and recursive customization, we consider that the full aspects of customizable services need to be described using a formalized language, and propose such a language called Web Service Variability Description Language (WSVL). WSVL defines not only what customization options are, but also how to construct customization messages and where to exchange such messages based on the XML technology. The language captures sufficient information so that consumers can customize services in a similar fashion to the usual service consumption without being restricted to a specific technology. In addition, the explicit description of variant service capabilities and enabled conditions help to capture variability inter-dependencies and allow customizable services to be consumed recursively.

To demonstrate the applicability of WSVL, we describe a reference architecture for service deployment and a service engineering process. While the reference architecture defines blueprint for enhancing existing service frameworks to provide customizable services, the engineering process describes how to use Model-Driven Engineering (MDE) techniques to engineer customizable services in a very
similar way to engineering conventional services. We also describe a prototype system which we have deployed to Amazon cloud to demonstrate the feasibility of our approach.

The rest of the paper is organized as follows. Section II presents a motivating example and outlines the use of feature models in enabling service customization. Section III describes WSVL with examples. The reference architecture for deploying customizable services is explained in section IV, while section V details the process for engineering WSVL-based services. We then present our prototype system in section VI. Section VII discusses related work and we conclude the paper in section VIII.

II. MOTIVATING EXAMPLE AND BACKGROUND

A. Motivating example

Swinsure Insurance is a wholesale insurance company providing building insurance service to various consumers (i.e. insurance brokers). In this paper, we use two terms, consumers and brokers, interchangeably to denote players who directly customize and invoke the Swinsure Insurance Web service (namely Swinsure WS). They are different from individuals (generally referred as end-users) who request quote and purchase insurance policies through brokers’ channels. After analyzing consumer requirements, Swinsure Insurance identifies the following variability in its capability which is captured by a feature model in Figure 1:

- Some (but not all) brokers need to view and/or update policy (Features 2, 4).
- All brokers need to support policy quoting but only some wish to enable policy purchasing as well (Features 3, 5, 6).
- All brokers only support one of two policy types, namely residential insurance and business insurance (Features 7, 10, 11).
- Brokers are able to add extra cover (i.e. accidental damage, fusion cover and extended third party liability) (Features 8, 12, 13, 14).
- Regarding payment method, some brokers wish to support credit card payment while others prefer to issue invoice payment (Features 9, 15, 16).

In addition, these variations cannot be arbitrarily combined but there are constraints among them:

- Those brokers who need to support policy update always need to enable policy query.
- The extra cover “extended third party liability” is only available to business insurance policies.

B. Feature modeling

Feature models are used in SPL for capturing the commonalities and differences among a family of software products [9]. While there are many manifestations of feature model since its original proposal [10], we utilize the Cardinality-Based Feature Modeling (CBFM) technique proposed by Czarnecki [11]. This technique provides a complete semantics for capturing service variability.

A feature represents a business functionality encapsulating broker requirements. A feature model is a hierarchy of features with composed-of relationships. For instance, the root feature “Swinsure Insurance Web Service” is composed of three features: one mandatory feature “Policy Creation” and two optional features “Policy Query” and “Policy Update”. The optionality of a feature is defined by the property “feature cardinality” represented above it. Feature cardinality determines the lower and upper bounds for a feature to appear in one feature configuration. A feature configuration is produced from the feature model by resolving all variability in the model. It corresponds to the requirement set of one broker. Group cardinality is shown below a feature to limit the number of child features in one feature configuration when the parent feature is selected. For instance, if the optional feature “Extra Cover” is selected, at least 1 and at most 3 of its child features (i.e. features 12, 13, 14) can be selected. In addition to feature hierarchy, there are cross-tree constraints which describe the inclusive and exclusive dependencies among features (cf. the constraint box). Our feature modeling tool [5] shows that the example feature model has 108 valid feature configurations. Consequently, there are 108 variants for the Swinsure WS.

Developing a customizable service will increase efficiency in service description, publication, and consumption.

C. Feature-based service customization

To reduce the complexity in customization process, we have proposed a feature-based service customization framework [5]. Specifically, features models are used to capture service variability at the requirement level. Consumers can customize the service by selecting necessary features and disabling irrelevant features. Such selections result in feature configurations which are used as customization requests from which the provider produces appropriate service variants. Figure 2 presents a screenshot of our consumer prototype tool which consumers can use to customize services as described above.

The advantages of using feature models are twofold. Firstly, service variability is captured at a high level of abstraction, thus helping to reduce the number of customization options. Secondly, dependencies among customization options can be formally captured as cross-tree constraints, thus enabling automated validation [12].
III. **WEB SERVICE VARIABILITY DESCRIPTION LANGUAGE (WSVL)**

WSVL is an extension of WSDL for describing customizable services. The language is built upon the concept of feature-based service customization. It enables the loose coupling because there is no restriction on the technology that consumers have to use. Instead, consumers are able to customize a service in a similar way to conventional service consumption. In addition, the language explicitly captures all information that consumers need to know to perform customization. WSVL also facilitates the capturing of service variability inter-dependencies to enable recursive service customization in service ecosystems.

The language specifies XML notation for capturing four pieces of information: service variability, customization endpoint, service capability, and feature mapping. While the key concepts of WSVL are explained here, more detailed explanation and its usage can be found from our research project website [13].

A. **Description of service variability**

This part specifies “what customization options are” Since we advocate feature-based service customization, the description of service variability is actually the XML representation of the feature model. Hence, this description contains two pieces of information: feature hierarchy and feature constraints. Regarding feature hierarchy, we use containment relationships among XML elements to represent composed-of relationships in feature model and two properties (i.e. minCardinality and maxCardinality) to specify feature and group cardinalities. Feature constraints are represented as logical expressions of elements which refer to features in the feature hierarchy using XPath expressions. Figure 3 is a snippet of the service variability description for the motivating example.

B. **Description of customization endpoint**

This part plays the key role in enabling the loose coupling between providers and consumers. In general, it defines two things: “how to construct customization messages” and “where to exchange those messages” The description specifies a set of customization operations defining what kind of customization requests are accepted by the providers and what will be the corresponding customization responses. In addition, the description contains information about mapping such customization messages to transport and messaging protocols, and information about endpoints where customization messages can be exchanged.

![Figure 2 - Feature-based service customization](image)

**Figure 2 - Feature-based service customization**

**Figure 3 - Description of service variability**

In each customization operation, the request contains a set of variant features representing consumer decisions, while the response contains information related to the resulting customized service. While customization operations are application specific, typical ones are:

- The request contains a feature configuration and the response contains the WSDL of a service variant.
- A variation of the above operation in which the response contains a URI of service variant’s WSDL.
- The request contains an incomplete feature configuration, and the response contains an URI of the WSDL for a customizable service variant. This usage enables multi-stage service customization, as well as the brokering of customizable services.

Generally, customization endpoint description defines a specialized service interface for management purposes. Hence, it is possible to reuse existing WSDL notation to represent this information. However, a WSVL description also contains information expressed by WSDL notation (such as service capability description as explained later). A good design principle is to make explicit which information is for customization endpoint and which information is for conventional service description (i.e. separating customization parameters from service consumption parameters). Therefore, we define new XML notation which inherit WSDL notation for this specific purpose.

![Figure 4 - Description of customization endpoint](image)

**Figure 4 - Description of customization endpoint**

Figure 4 presents the customization endpoint description for SwinSure WS. The snippet defines one customization operation, namely “customizationOperation”. This operation accepts a set of enabled features and a set of disabled features and returns the URL of the service variant’s WSDL.
Due to space limitation, we omit details of message definitions. The binding element, "<wsvl:binding>" specifies that input and output messages of this operation will be formatted as SOAP messages and transported using HTTP protocol. In addition, the element "<wsvl:port>" defines the endpoint at which customization messages are exchanged.

C. Description of service capability and feature mapping

The service capability description captures the superset of abstract capability (i.e. portTypes, operations, data types) of all service variants, while the feature mapping description captures the correspondences between variant features and variant capabilities. These descriptions collectively define what capability is available for a given feature configuration without the need to actually customize the service. Consequently, it enables the capturing of service variability inter-dependencies.

We utilize WSDL notation for the service capability description. Figure 5 presents a snippet of this description for the Swinsure WS. There are two portTypes, namely “quotingPortType” and “purchasingPortType”. And the “quotingPortType” has 4 operations.

Regarding the feature mapping description, the availability of a variant service capability is generally decided by the combination of two sets of features. In particular, the inclusion of features in the first set and the exclusion of features in the second set collectively result in the availability. In WSVL, each correspondence is represented by a set of links and one link specifies one mapping between a variant feature and a variant capability. A link has additional attributes specifying whether the inclusion or exclusion of the feature decides the availability and the type of the capability (i.e. portType or operation).

Figure 6 presents the mapping description for the Swinsure WS. The first two links define that the operation “getQuote4Residential” of the portType “quotingPortType” (cf. Figure 5) only exists in a service variant if the feature “Residential” is enabled and the feature “ExtraCover” is disabled. Otherwise, it is excluded from variants. Similarly, the last link defines that the portType “purchasingPortType” only exists if the feature “Purchase” is enabled.

To explain how WSVL helps to support variability inter-dependencies, let us consider a scenario in Figure 7. Swinbroker is a consumer that provides an insurance quoting business process. This business process is also customizable and its users can select one among three variants: residential policy without extra cover, business policy without extra cover, and business policy with extended third party liability. Therefore, Swinbroker business process has a general task, namely “Quoting”, which is mapped to one of three operations provided by Swinsure WS depending on user selection. The description of the service capability and the feature mapping of Swinsure WS allow Swinbroker to capture these dependencies at the process modeling level without the need of actually customizing Swinsure WS. And when a user chooses one of its variants, Swinbroker will dynamically customize Swinsure WS to get a particular service variant with an appropriate operation to invoke.

IV. REFERENCE DEPLOYMENT ARCHITECTURE

In the provisioning of WSVL-based customizable services, there are three common capabilities which are shared among all services from the same service provider:

1. The validation of the requested feature configuration. The requested feature configuration needs to be validated against the corresponding feature model. For instance, in case of Swinsure WS, it is a fault if a consumer requests a service variant with both features “Residential” and “Extended Third Party Liability” enabled due to their excluded relationship.

2. The dynamic derivation and deployment of service variants. Given a feature configuration, the provider needs to dynamically derive and deploy a variant whose capability is a subset of the service capability.

3. The management of the lifecycle of service variants. Customizable services need to keep track of what variants are already deployed. The records help manage the availability of service variants, as well as reusing existing ones if incoming requests contain the same feature configuration as the ones already processed.

To facilitate service providers in providing a suite of customizable services, we propose a reference architecture for service deployment (cf. Figure 8). The reference architecture separates capabilities common to all customizable services from capabilities specific to each service. Consequently, it eases the development of new customizable services.
For each customizable service, there are two endpoint types: customization endpoint and variant endpoint. The customization endpoint is uniquely specified in the WSVL document for exchanging customization messages, while variant endpoints are dynamically created for exchanging messages related to consuming particular service variants. In general, a new variant endpoint is created each time a new feature configuration is requested. Other components and their interactions are described below using the scenario when there is a new incoming customization request.

i. The **Customization frontend** is a service for providing customization operations. Its capability is specified in the customization endpoint description. When the service accepts a customization request, it queries the **Feature model management** for the validation of the requested feature configuration. If the feature configuration is invalid, an error message is returned. Otherwise, it is passed to the customization engine.

ii. The **Feature model management** is for storing feature models and validating feature configurations.

iii. The **Endpoint repository** keeps track of where existing variants are deployed. It also contains deployment status (e.g. active, expired) for managing variant lifecycles.

iv. The **Configurable service implementation** is the implementation of the service capability. Providers can exploit any technique for this component as long as it is configurable by the customization engine.

v. For each feature configuration, the **Runtime customization and management engine** first queries the **Endpoint repository** to check if a variant is already deployed and is still active. If yes, the engine returns the corresponding variant endpoint to the **Customization frontend**. Otherwise, the engine is going to derive a new service variant by configuring and deploying the **Configurable service implementation**.

vi. When a new variant is deployed, the new deployment endpoint (i.e. variant endpoint) is registered to the **Endpoint repository**, as well as being returned to the **Customization frontend**.

It should be noted that, only the component **Configurable service implementation** is specific to one customizable service. While the **Customization frontend** needs to be uniquely deployed for each service, its operations as well as operations of other components are common to all services. In other words, to provide a new customizable service, providers only need to develop a configurable service implementation and reuse other components for the deployment and provisioning of the service.

V. **SERVICE ENGINEERING**

In this section, we describe a model-driven engineering technique for developing **Configurable service implementation**. In addition, we present how to construct the WSVL document for the corresponding service.

A. **Overview**

Figure 9 presents the engineering process for developing a configurable service implementation. Firstly, CBFM is used to model the variability of the customizable service to produce a feature model. Secondly, the abstract capability of the service as well as the mapping between the abstract capability and the feature model is constructed. Thirdly, a configurable service implementation is developed. Due to space limitation, we omit the description of the first step whose details can be found in [5].

B. **Service model and feature mapping model**

To model service capability, we develop a metamodel (cf. Figure 10) based on Marcos’ UML notation for WSDL [14]. In particular, a service is comprised of one or more interfaces which in turn consist of one or more portTypes. Each portType defines one or more operations. An operation specifies types of messages for input, output, and fault.

The service model represents the superset of capabilities of all service variants. The presence of some variant capabilities depends on particular features required by consumers. To capture the correspondence between variant features and variant service capability, we utilize a feature mapping metamodel (cf. Figure 11). A **MappingModel** relates features in a feature model, referenced by **FeatureModelRef**, with service elements in a service model, referenced by **ServiceModelRef**. It is composed of **Links** and each **Link** consists of a **Feature** and at least one **ServiceElement** which reference elements in the feature model and the service model respectively. It should be noted that the service model and the feature mapping model are
semantically equivalent to the description of service capability and feature mapping in WSVL.

![Feature mapping metamodel](image1)

C. Configurable service implementation

Java API for XML Web Services (JAX-WS) is a widely-used standard for engineering services [15]. JAX-WS defines mapping between elements of a service interface description and service implementation code (e.g., interfaces, classes or methods) so that service skeleton can be automatically generated from service interface description. From this skeleton, service developers only need to complete the business logic to have an executable service. The key mapping rules in JAX-WS are:

i. A portType element is mapped to a Java interface.

ii. An operation element is mapped to a method within the corresponding interface.

iii. XML schema types for message definitions are mapped to Java objects.

JAX-WS uses annotations for specifying element mapping so that service engines are aware of the exposing service interface. For instance, the portType “quotingPortType” defined in Figure 5 will be mapped to a Java interface with the “@WebService” annotation (cf. Figure 12). Similarly, two operations “getQuote4Residential” and “getQuote4ResidentialWithExtra” are mapped to two methods with the “@WebMethod” annotation.

To develop configurable service implementation, we extend JAX-WS to support variation points (or configuration points) within the service skeleton. To this end, there are two types of variation points that need to be considered. Firstly, there are variation points in the code structure reflecting the variability defined in the service model. For instance, Java interfaces realizing alternative portTypes will become alternative, while Java methods implementing optional operations will become optional. Secondly, there are variation points within the implementation logic of such methods. For instance, assuming that the method “getQuote4ResidentialWithExtra” is to produce a quote for residential policy with extras, its logic needs to consider which extras (i.e., “Accidental Damage”, “Fusion Cover” and “Extended Third Party Liability”) are enabled.

To support the first variation point type, we define a new JAX-WS annotation type, namely FeatureMapping. Examples of the new annotation type are demonstrated in Figure 12 with italic font. Similar to the semantic of mapping models, each variant capability will be annotated with two sets of features. The first set contains features whose selection results in the existence of the capability (i.e., enabledFeatureList property), while the exclusion of features in the second set contributes to the existence of the capability (i.e., disabledFeatureList property).

![JAX-WS annotation example](image2)

The second type of variation points is application specific. There is no general relation between these variation points and variability specified in the WSVL document. Therefore, we exploit the concept of Web Service Context in JAX-WS for supporting this category. Web Service Context is a Java resource injected at runtime by service engine and can be referred to in the implementation logic of Web services. Thus, we use this property to initialize the service context (i.e., which features are enabled/disabled) for a particular service variant at the time the variant is deployed. In the business logic of the service implementation, such information can be referred to in order to realize variability.

The use of Web Service Context extension is demonstrated in Figure 13 which simulates the implementation logic of the method “getQuote4ResidentialWithExtra”. The variable context is injected by the service engine at runtime and contains initialized feature set for one variant. Consequently, the first code block is only executed if the feature “Accidental Damage” is enabled, while the second code block is controlled by the feature “Fusion Cover”.

![Web Service Context extension](image3)

D. Generation of WSVL document

As explained in section III, a WSVL document contains 4 different pieces of information. Three pieces contain information which is semantically equivalent to models developed during service engineering. In particular, the description of service variability (cf. Figure 3) captures information from the feature model (cf. Figure 1) while the description of service capability and feature mapping is XML representation of the service model and the feature mapping model. Therefore, models produced during the service engineering are used to automatically generate corresponding information in the WSVL document. Once the service Customization Frontend (cf. Figure 4) is specified and developed by providers, its WSDL description is added to above generated information to produce a complete WSVL document of the corresponding customizable service.
VI. PROTOTYPE IMPLEMENTATION

We have developed a prototype of our reference architecture on Apache CXF [16], an open source Web service framework which fully supports JAX-WS. The prototype is currently running on the Amazon Cloud [13]. The prototype enables service providers (e.g. Swinsure Insurance) to develop and deploy WSVL-based customizable services as follows.

To develop the Configurable service implementation, a set of Eclipse plugins have been implemented using the EMF. These plugins enable the development of service engineering models (see [5] for more details). From these models, we use XSLT transformation to generate the description of service variability, service capability and feature mapping following WSVL XML Schema. We then use WSDL2Java tool, which is an implementation of JAX-WS, to generate service skeleton from the description of service capability. Variant capability in the service skeleton is then annotated using the new annotation type (cf. Figure 12) and the implementation logic of the service with built-in variation points is constructed (cf. Figure 13).

![Figure 14 – WSVL^1 and WSDL^2 retrieved from the prototype tool](image)

To enable the dynamic deployment of service variants, we use Java reflection [17] in implementing the Runtime customization and management engine. In particular, given a feature configuration, the engine examines the new annotation type (i.e. FeatureMapping) in the Configurable service implementation and decides if the corresponding variant capability should be exposed in a service variant. The engine also initializes the service context before deploying the service variant. Other components of the reference architecture are implemented as stand-alone components to enable reuse. In particular, the Customization frontend exposes WSVL documents while the Feature model management extends the FAMA framework [18].

To validate the prototype, we have also implemented and deployed Swinsure WS on it. Figure 14 presents the screenshots of using Swinsure WS. Its WSVL description^1 is shown in the top-left window, while the WSDL description^2 for the requested variant in Figure 2 is shown on the bottom-right window. The dynamically deployed variant has only 1 portType and 4 operations while the original service has 2 portTypes and 11 operations. By invoking the operation “getQuote4ResidentialWithExtra” of this variant, we confirmed that its behavior takes into account that only feature “Accidental Damage” is enabled while other two child features of the feature “Extra Cover” are disabled.

The consumer prototype tool (cf. Figure 2) for interacting with Swinsure WS is available from our project website [13]. The tool can be used as an Eclipse plugin or from console to test the capability of Swinsure WS as explained above.

VII. RELATED WORK AND DISCUSSION

There are a number of works for enabling service customization in literature [3, 4, 8]. In [3, 8], the authors use EMF to capture variation points and variants, as well as their dependencies at the technical level of service description (i.e. operations, messages, data types). Consumers are then required to use EMF to select appropriate variants. Due to the explicit use of a specific technology (i.e. EMF) and the missing procedure for exchanging customization information, the approach is only applicable when both providers and consumers belong to the same organization. In [4], the authors extend the WS-Policy framework to describe customization options and check the validity of customization requests. The approach presumes that consumers know how to format and where to exchange customization messages. In addition to the complexity incurred by the customization process, these approaches enforce the tight coupling between providers and consumers. Another thread of works proposes techniques for modeling service variability, e.g. [19, 20]. While these works contribute to the identification of variability in service interface description, they do not define mechanisms for enabling service customization.

There are also related works from perspectives of variability description languages for services and service-based systems, e.g. [21-24]. However, all approaches, except Web Service Offering Language (WSOL) [21], address the description of variability in business processes to enable variant derivation within an organization. This is not the focus of this paper since we aim to support customizable atomic services. WSOL aims to describe different classes of a Web service whereas each class is determined by a combination of constraints (e.g. QoS, access rights). While WSOL enables the selection of service variants by focusing on non-functional properties, our work focuses on functional properties. Since both languages are compatible with WSDL, we believe they are complementary in enabling the complete scope of service variation.

Our technique for engineering customizable services is influenced by techniques from SPL for feature-oriented product line engineering and variability implementation, e.g. [25-30]. As explained, our solution extends JAX-WS. While

---

2 Service variant’s WSDL: http://ec2-107-21-175-19 compute-1.amazonaws.com:8080/swinsureServiceVariant/1/Quoting?wsdl=1_full
there may be other ways for achieving variability in the service implementation, we adopt this technique due to its widespread use. In addition, providers are encouraged to use any in-house technique for developing the configurable service implementation which will still be deployable using our deployment architecture and be exposed using WSVL.

One might argue that, our solution necessitates the use of the feature modeling technique which is one type of technology that enforces the tight coupling between providers and consumers. However, our argument about loose coupling is not contradictory due to three reasons. Firstly, the introduction of the feature modeling technique is essential to address the complexity in the service customization process. Secondly, feature models are semantically interpreted as propositional logics which can be reasoned and validated using variety of existing solvers [12]. Hence, there is no such a requirement that providers and consumers have to use the same tools for reasoning. Thirdly, the loose coupling is not only about technology specificity, but also about implicit agreement for which we proposed WSVL to formalize the customization interfaces.

VIII. CONCLUSION

In this paper, we have described a language, called WSVL, and its application to support the delivery of customizable Web services. WSVL captures full aspects of customizable services to enable loose coupling between providers and consumers, as well as allowing customizable services to be consumed recursively. The language is an extension of and is compatible with WSDL that helps to align WSDL-based customizable services with WSDL-based services. Consequently, it facilitates the introduction of customizable services as first-class entities in SOA. We introduce a reference architecture for service deployment which is a useful blueprint for extending existing service platforms to support customizable services. The service engineering process demonstrates how the development of customizable services can be supported by MDE techniques in a similar fashion to conventional service development. While this work focuses on atomic services, in future work, we plan to address challenges in integrating WSVL with the provisioning of customizable composite services.

ACKNOWLEDGMENT

This research was carried out as part of the activities of, and funded by, the Smart Services Cooperative Research Centre (CRC) through the Australian Government’s CRC Programme (Department of Innovation, Industry, Science and Research).

REFERENCES

6. E. Project, Eclipse Modeling Framework (EMF), accessed on 30 April 2012