A Feature-Based Framework for Developing and Provisioning Customizable Web Services

Tuan Nguyen, Alan Colman, and Jun Han

Abstract—A customizable Web service is a service that enables service consumers to dynamically determine variants of the service they receive. Provisioning customizable services helps to efficiently address functional variability in customer requirements. However, this is challenging due to: i) the complexity in deriving the right subset of service capabilities for a service variant and ii) the existence of a large number of variants and their dependencies. We propose a feature-based framework to tackle this challenge. In our framework, a feature model is used to capture functional variability in customer requirements at a high-level of abstraction and to provide customers with a much simpler way to customize an atomic service. A service engineering process is designed to facilitate the systematic identification and implementation of variability during service development, and to maintain the mapping between variabilities at the feature modeling level and the service implementation level. We define a generative middleware that supports service deployment and exploits the mapping to enable runtime service customization. A large scale case study based on the Amazon Web Services is used for evaluation. In addition to addressing the challenge in provisioning customizable services, our experiments show that the generative middleware helps to reduce runtime resource consumption.

Index Terms—Web services, Service customization, Feature modeling, Service middleware

1 INTRODUCTION

Along with the proliferation of Web services technologies, providers of complex Web services are facing a significant challenge arising from variability in customer requirements. There are two types of variability: functional and non-functional variabilities. The former one refers to different service capabilities required by different customers who share a set of core service capabilities. For the latter one, all customers require the same set of service capabilities but each of them has different requirements in the quality of the provisioned service. In this paper, we particularly focus on functional variability.

Variability in functional requirements results in a large number of service variants with slightly different service capabilities. When leveraging existing Web service technologies, service providers have two alternatives to address this challenge. The first option is for service providers to prepare a variant of the Web service for each unique set of customer requirements. We refer to this approach as a multiple-variants approach. Alternatively, service providers can use a super-service approach. A super-service is a single instance that provides all service capabilities required by all customers.

Each approach has its own advantages and disadvantages. The multiple-variants approach has the advantage of having a lean service interface which is tailored to specific customer requirements. However, the disadvantage of this approach is that each service variant needs to be individually developed and maintained. As the number of service variants increases, managing multiple deployments and updating shared functionalities becomes increasingly onerous.

In contrast, the super-service approach has the advantage of only having one service that needs to be developed and maintained. However, from any particular customer’s perspective, the service interface of the super-service contains redundant service capabilities. This redundancy not only makes it more difficult to understand a service, but also results in a larger memory footprint and unnecessary “boilerplate” code in the customer business applications that act as service clients.

In addition, the selection of a service variant in multiple-variants approach or the selection of a compatible subset of service capabilities in super-service approach that suits a particular customer’s needs is a complex process. In either case, customers need to justify their selection from a large set of service variants or service capabilities based on the service description. Such description is usually based on a low-level technical description of service capabilities rather than a high-level description of their business objectives, thus making the selection process complex. Also, there are often dependencies among different service capabilities (e.g., the use of one service capability requires or excludes other service capabilities) which are given in natural languages and are not amenable to enforcement.

Managing variability in service development and provisioning is also a challenging task for both approaches. As the number of differences between customer requirements increases, the number of variation points1 in service im-

---

1 A variation point is a place in the service implementation where the service logic switches among different alternatives depending on the requesting customers. For instance, a service might load different components for exposing different service capabilities.

---
plementation increases greatly. Furthermore, dependencies between the variants\(^2\) bound to these variation points become much more complex. Precisely implementing such variability and switching between alternative behaviours is far from a straightforward task.

In order to tackle the shortcomings inherent in both the multiple-variants and super-service approaches, the use of customizable services is promising. A customizable service is associated with a formal process through which customers are able to define and generate the exact service variants that they wish to consume. These service variants have concise service interfaces which are dynamically tailored to consumer requirements. In addition, they all share the same single service implementation, thus facilitating service maintenance. To completely address variability in consumer requirements, such a service customization approach needs to tackle the issues related to the complexity in selecting a compatible set of service capabilities and variability management.

In this article, we present a feature-based framework for developing and provisioning Web services that can be customized through feature selection. This framework embodies the principles of feature modeling [1] and generative programming [2] from the Software Product Line (SPL) literature [3]. Features are a natural abstraction of similarities and differences among customer requirements. A feature model is a hierarchical organization of features with cross-tree constraints. In our framework, feature models provide a convenient way of organizing customer requirements at a high-level of abstraction. By mapping variability in features to variability in the service interface and then in the service implementation, the framework also enables the automatic generation of service variants based on customers’ feature selections.

Our framework comprises a feature-based service customization process, an SPL-based service engineering process, and a generative middleware for service deployment and provisioning. The service customization process helps to reduce complexity in selecting a compatible set of service capabilities in two ways. Firstly, it provides an automated constraint enforcement mechanism that frees customers from needing to ensure the compatibility of selected capabilities. Secondly, it abstracts away the technical description of service capabilities and allows customers to focus on the business objectives (i.e., features) that a service variant can provide.

The service engineering process provides a systematic way of identifying and managing variability throughout the development of customizable services as well as during service provisioning. In addition, the generative middleware extends conventional service middleware to incorporate customizable services. It enables the deployment of customizable services as well as the automated derivation of service variants.

This paper only focuses on customizable atomic services. This work is complemented with our related work that addresses issues in supporting customizable composite services and dynamic variability resolution [4, 5].

In addition, this work is a significant extension of our previous work [6] which describes an interface description language for customizable Web services. The new contributions of this paper relative to [6] are as follows:

- We provide a detailed account of the service engineering process and present a systematic way of modeling and managing variability;
- We define a variability resolution mechanism for the generative middleware to support runtime service variant derivation and provisioning;
- We perform a large-scale evaluation based on a commercial complex Web service. Our experiments show that for customers of light-weight or medium-weight requirements, a significant reduction of runtime resource consumption is achieved as the result of customizing and dynamically deploying service variants.

The rest of the article is organized as follows. Section 2 introduces an expository case study from the insurance industry as well as the relevant research background. An overview of our feature-based service customization framework is presented in Section 3. We describe the software engineering techniques for developing customizable services in Section 4. Section 5 discusses our generative middleware for service deployment and runtime service provisioning. We present the prototype implementation in Section 6 and the evaluation of our framework in Section 7. Section 8 discusses related works and Section 9 concludes the paper.

2 CASE STUDY AND FEATURE MODELING

2.1 Swinsure Insurance Case Study

The case study describes an insurance quoting service provided by Swinsure Insurance. Swinsure Insurance is a wholesale building insurance provider that provides building insurance through various insurance brokers. Customers of Swinsure Insurance range from light-weight brokers, who only need to give end-users building insurance quotes, to medium-weight ones, who also provide end-users options on purchasing insurance policies, to heavy-weight ones, who offer full-fledged insurance services (e.g., providing various extra covers).

Swinsure Insurance develops a Web service, called Swinsure Insurance Web service, for supporting the integration with brokers’ business applications. Due to the differences in broker requirements, there is variability (i.e., commonalities and differences) in the respective capabilities of this Web service needed by each individual insurance broker. In addition, there are dependencies among variant service capabilities. For instance, the extra cover “extended third party liability” can only be bundled with business policies not with residential policies. In the following section, we introduce the concept of feature modeling and demonstrate how the technique can be used to capture this variability.

2.2 Feature Modeling

Feature models are key elements for identifying and managing variability in Software Product Line engineer-

\(^2\) A variant in the service implementation is one possible business logic implementation that can be bound to a variation point.
This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TSC.2015.2405546, IEEE Transactions on Services Computing

T. NGUYEN ET AL.: A FEATURE-BASED FRAMEWORK FOR DEVELOPING AND PROVISIONING CUSTOMIZABLE WEB SERVICES

Fig. 1 - A feature model for Swinsure Insurance Web service

starts from the root feature which is always selected. As the result, the process produces a feature configuration which is actually made of a set of selected features and a set of disabled features.

2.3 Software Product Line and Generative Programming Paradigm

Software Product Line is a software engineering paradigm that seeks to exploit commonalities among systems from a given problem domain while systematically managing the variability among them. SPL clearly separates two processes of software development: domain engineering process and application engineering process [3].

The domain engineering process defines the commonality and the variability of a product line, as well as constructing reusable assets that accomplish the desired variability. The domain engineering process also defines mechanisms by which variability is resolved.

In contrast, the application engineering process is concerned with the derivation and development of particular products from reusable assets. The product’s specific requirements are taken into account and the variability mechanisms defined in the domain engineering process are followed in order to develop the product in a timely manner with the least cost and highest quality.

The generative programming paradigm is an SPL approach which focuses on automating the creation of particular products from reusable assets [2]. The key concept is to maintain the mapping between variability in the problem domain and variability in the solution domain so as to enable this automated product derivation. The format of this mapping is subject to the particular application domain involved.

3 OVERVIEW OF FEATURE-BASED SERVICE CUSTOMIZATION FRAMEWORK

3.1 Inspiration from SPL

There is a strong similarity between a product family and a customizable service. Firstly, both represent a set of applications (or service variants) which share common capabilities. And each application requires some optional capabilities which help differentiate one application from another. Secondly, they both need to explicitly define and manage variability so that the resolution of such variability will help to derive applications tailored to specific customer needs. Thirdly, the development of customizable services is analogous to the domain engineering process, while service customization is similar to the application engineering process.

A feature model provides an efficient way to capture and communicate service variability at the highest level of abstraction (i.e., the requirement level). A feature model provides customers a succinct view of what can vary. Also, it helps customers to focus on the business objectives (i.e., features) that a customizable service can provide, rather than the technical details (e.g., message and data type definitions) related to those objectives. A feature model also has a built-in mechanism to capture and enforce constraints which helps customers to make in-
formed decisions during the customization process.

We propose the use of a feature model as a service description artefact for representing customization options, while a customizable service is developed as a product family driven by variability in the feature model. Consumers customize the service by deciding the features they want (i.e., selected features) and features they do not want (i.e., disabled features). This is similar to configuring a feature model to derive a feature configuration. This feature configuration becomes customers’ customization decisions according to which a service variant is automatically derived and deployed from the feature-oriented product family.

In the following sections, we present an overview of customizable services in operation based on this idea. In particular, we describe how customizable services are projected from customers and providers’ viewpoints respectively.

3.2 Customer View of Customizable Services

A customizable service operates in two modes: customization mode and consumption mode (Fig. 2). The customization mode involves activities in which customers comprehend the customization options (i.e., feature model) and make customization decisions (i.e., create a feature configuration). In addition, this mode accommodates interactions for exchanging customization requests and responses. A customization request is automatically processed by the customizable service so as to dynamically derive a service variant. This service variant is communicated to the customer as part of the customization response. As described earlier, feature models play a significant role during the customization mode for both representing customization options and deriving service variants.

In the consumption mode, a customer has access to or can use a particular service variant. The service variant is a conventional service which exposes the exact service capabilities that the customer requires. Other service capabilities required by other customers with different requirements are not accommodated by this service variant. Consequently, the customer does not have to worry about those irrelevant service capabilities. Note that the consumption mode is the only mode under which a conventional Web service operates.

3.3 Provider View of Customizable Services

Customizable services are developed as a feature-oriented product family by following SPL engineering principles and the generative programming paradigm. In particular, during the domain engineering process, we use a feature model to capture service variability at the requirement level as described in Section 2.2. This feature model drives the identification and the introduction of variation points and variants in reusable assets for service modeling and service implementation. As such, a feature model becomes a high level representation of a customizable service with all customization options. Also, the domain engineering process produces reusable assets as configurable artefacts (cf. Fig. 3). Following the generative programming paradigm, we maintain the mapping (i.e., correspondence) between variability in these configurable artefacts and variability in the feature model.

During the application engineering process, the feature model facilitates the automated derivation of particular service variants. In particular, the requirement set of a customer is captured as a feature configuration. Exploiting the mapping between the feature model and configurable artefacts, our generative middleware automatically resolves the variability within those configurable artefacts. This step yields necessary artefacts required for a particular service variant. Also, the generative middleware dynamically deploys those artefacts so that the service variant becomes available to the requesting customer in the consumption mode.

4 Domain Engineering Process for Customizable Services

4.1 Overview

Fig. 4 presents an overview of the domain engineering process. The process comprises two main phases: i) service capability and variability modeling, and ii) service implementation and variability realization. In Fig. 4, greyed boxes represent activities for engineering conventional Web services. Other boxes are added by our framework to engineer customizable Web services. As such the domain engineering process for customizable services is a direct and significant extension of the conventional service engineering process that takes into account issues related to varia-
4.2 Service Capability and Variability Modeling

This phase consists of three major engineering steps. Firstly, variability in customer requirements is captured using the feature modeling technique. Secondly, developers will design the service interface, defining common and variant service capabilities according to variability identified at the feature level. Thirdly, the correspondence between variability in the feature model (i.e., variant features) and variability in the service capability model (i.e., variant service capabilities) is captured by means of a feature-capability mapping model. In the following subsections, we explore each of these steps in more detail.

4.2.1 Feature Modeling

A feature model is instantiated from the CBFM (Cardinality-Based Feature Modeling) metamodel [7]. Fig. 5 illustrates the feature model and its constraints for the Swinsure Insurance Web service with a screenshot of our feature modeling tool. This tool extends an open source tool (Eclipse Modeling Framework) with new functions related to service customization. In this tool, constraints among features are propositional formulas associated with the root feature.

4.2.2 Service Capability Modeling

A service capability is either a service operation or a portType that represents functionalities of a service. Service capability is modeled using modeling notations similar to WSDL elements (e.g., service, portType, operation). The service capability model for a customizable service accommodates capabilities of all service variants. During the customization process, only the capabilities matching the consumer requirements are retained while other irrelevant capabilities are purged to produce capabilities of one particular service variant.

Service capability is modeled in accordance with the feature model. There are two types of service capability: common service capabilities and variant service capabilities. The former are ones that are shared by all customers. As such common service capabilities realize common features in the feature model. On the other hand, the latter are ones that are only required by a subset of customers. Variant service capabilities realize variant features in the feature model.

Fig. 6 shows the service capability model for our motivating example as created in our service capability modeling tool. This tool is created as an Eclipse plugin using the EMF framework (Eclipse Modeling Framework). PortType “quotingsPortType” is a common service capability while portType “purchasingPortType” is a variant service capability which is only available for customers who select feature “Purchase”. Also all service operations within portType “quotingsPortType” are variant service capabilities. For instance, operation “getQuote4Residential” is for customers who select feature “Residential” and disable feature “Extra Cover”, while operation “getQuote4ResidentialWithExtra” is for customers who select both feature “Residential” and “Extra Cover”.

Variability in the service interface model is the refinement of variability in the feature model. For instance, two alternative variant features “Residential” and “Business” may be realized by two alternative service operations for getting quotes, such as service operations...

---

Fig. 4 - Overview of the service engineering process

Fig. 5 – Feature modeling tool and Swinsure Insurance feature model

Fig. 6 – The service capability model for Swinsure Insurance
“getQuote4Residential” and “getQuote4Business”. In Fig. 6, each service operation is further refined according to whether feature “Extra Cover” is selected. In particular, there are two service operations for getting residential quotes. Operation “getQuote4ResidentialWithExtra” is used when feature “Extra Cover” is selected, while operation “getQuote4Residential” is used when feature “Extra Cover” is disabled. Operation “getQuote4ResidentialWithExtra” may also be further refined according to specific extra cover to be included. The decision for refinement depends on the actual business context, the granularity of data captured in one service operation, as well as the service design principles. In our motivating example, we do not further refine this service operation. Instead, we identify the included extra covers and switch the processing logic dynamically in the implementation of the service operation (as discussed later in Section 4.3.3).

4.2.3 Feature-Capability Mapping Modeling

The feature-capability mapping model is used to specify the refinement relationship between variability at the feature level and variability at the service capability level. It defines how variability at the service capability level should be resolved as the result of variability resolution at the feature level. In particular, it defines the presence condition of variant service capabilities as a propositional formula of variant features which is expressed as “If the variant features A1, A2, …, An are selected and the variant features B1, B2, …, Bm are disabled then variant service capability Cx is available”. That is, the availability of a variant service capability is determined by the conjunction of two sets of features. The first set contains selected variant features, while the second set contains disabled variant features. When this condition is satisfied with respect to a customization request, the variant capability is available to a service variant. Otherwise, the variant capability is not available to the service variant.

Fig. 7 presents the metamodel for mapping between variant features and variant capabilities. A MappingModel consists of three element types: FeatureModelRef, ServiceModelRef, and Link. FeatureModelRef and ServiceModelRef are references pointing to the corresponding feature model and service capability model. Links are used to relate variant features in the feature model to variant capabilities in the service capability model. To this end, a Link consists of a Feature and a number of ServiceElements. A Feature has a boolean attribute “presence” to denote whether the selection (i.e., TRUE) or disabling (i.e., FALSE) of the feature contributes to the presence condition of the corresponding ServiceElement. In addition, a ServiceElement also has an attribute “target” to denote the type of the service element (e.g., portType or operation). It should be noted that the mapping between one ServiceElement and multiple Features is realized by multiple links each of which has its own presence attribute for the corresponding Feature. This set of links collectively defines the presence condition of the ServiceElement.

The mapping between features and capabilities is illustrated in Fig. 8. This is a screenshot from our feature-capability mapping tool which facilitates the instantiation of the feature-capability mapping models from the mapping metamodel. This tool is implemented as an extension of an open source Eclipse plugin, called Atlas Model Weaver (AMW) [9]. In the screenshot, the feature model is presented on the left panel, while the service capability model is presented in the right panel. The middle panel shows the mapping model. Two links are expanded in the screenshot to demonstrate the mapping with respect to the operation “getQuote4Residential”. While the first link references the feature “Residential” with the attribute “presence” set to TRUE, the second link references the feature “Extra Cover” with the attribute “presence” set to FALSE (as shown on the bottom of the figure). These two links collectively define that “Residential ∧ ExtraCover” is the presence condition for service operation “getQuote4Residential”.

![Feature-capability mapping metamodel](image)

Fig. 7 – Feature-capability mapping metamodel

![Feature-capability mapping model for Swinsure Insurance](image)

Fig. 8 – Feature-capability mapping model for Swinsure Insurance
4.3 Service Implementation and Variability Realization

Engineering conventional Web services using Model-Driven Engineering (MDE) is a widely-used technique and consists of two steps. First, the service skeleton is automatically generated from the service capability model. Second, the business logic of the service is implemented by extending the generated service skeleton. The engineering of customizable Web services requires two further steps to produce configurable service implementation. These two steps incorporate variability (aka. configuration points) into service skeleton and business logic implementation respectively.

4.3.1 Service Skeleton Generation

The automated generation of a service skeleton relies on a mapping between elements for describing the service interface and programming constructs for realizing the service implementation. Our framework employs the widely-used JAX-WS standard that defines the following mapping rules:

- A portType is mapped to a Java interface, namely Service Endpoint Interfaces (SEIs), with @WebService annotation;
- An operation is mapped to a method in the SEI with “@WebMethod” annotation;
- XML schema types are mapped to data type classes;

4.3.2 Realizing Variability in Service Skeleton

As the service capability model contains variant service capabilities, the generated service skeleton also contains variant Java interfaces and methods. Due to the one-to-one mapping between service interface elements and Java constructs, the presence conditions of these variant Java constructs is the same as the presence conditions of the corresponding variant service capabilities. As discussed in Section 4.2.3, a presence condition is specified by a set of enabled features and a set of disabled features.

To support configuration points related to this type of service variability, we define a new JAX-WS annotation type, called FeatureMapping (cf. Fig. 9). Each FeatureMapping annotation has two properties. While the enabledFeatureList property is defined as an array of enabled feature names, the disabledFeatureList property is an array for storing disabled features. FeatureMapping annotations are available to be examined at runtime as specified by the @Retention annotation. This is important to enable the runtime engine to resolve variability dynamically.

Fig. 10 demonstrates how the FeatureMapping annotations are used in-line with JAX-WS annotations in the service skeleton. The method “getQuote4Residential()” is mapped to the service operation “getQuote4Residential” through the annotation “@WebMethod” (line 3). In addition, the FeatureMapping annotation (line 4) indicates that this is only available when the feature “Residential” is selected and the feature “Extra Cover” is disabled. Similarly, the interface “PurchasingPortType” is mapped to the portType “purchasingPortType” through the annotation “WebService”. The FeatureMapping annotation on line 8 defines that this interface is only available when the feature “Purchase” is selected. As demonstrated, FeatureMapping annotations represent the exact information specified in the feature-capability mapping model.

4.3.3 Business Logic Implementation

The implementation of business logic is performed by defining service implementation classes which implement SEI(s). Specifically, the business logic for each service operation needs to be defined in the method that represents it. For customizable Web services, there may exist variability within this business logic implementation that needs to be taken into account. We discuss this variability in the next section.

4.3.4 Realizing Variability in Business Logic

Variability in the business logic represents variability in the implementation logic of each variant service capability. For instance, the method “getQuote4ResidentialWithExtra” demonstrated in Fig. 10 is used to produce a quote for a residential insurance policy with extras. The method is available in all service variants providing any combination of three extras: “Accidental Damage”, “Fusion Cover” or “Extended Third Party Liability”. However, the actual quote depends on which extra covers a service variant provides. Therefore, the business logic of this method exposes variability with respect to the combination of extra covers provided in a particular service variant.

Variability in the business logic needs to be resolved at runtime by the runtime engine. There are a number of techniques for this purpose [10]. For instance, alternative behaviours are collected into separate classes/methods and a Design Pattern (e.g., Strategy, Template Method, or Abstract Factory) is used to select between these classes. Or alternative behaviours are dynamically injected at runtime using reflection and aspect-oriented techniques. In general, the use of runtime approaches requires the ability to dynamically switch between different processing logics. Therefore, we exploit the concept of Web Service Context in JAX-WS for supporting this type of variability. Web Service Context is a resource injected at runtime by the service engine and can be referred to in the implementation logic of Web services. Thus, we use this property to ini-

Fig. 9 - New JAX-WS annotation type

Fig. 10 - Example of the new JAX-WS annotation
Centralize the service context (i.e., which features are enabled/disabled) for a particular service variant at the time the service variant is deployed. In the business logic of the service implementation, such information can be referred to in order to realize variability. With this extension, developers are free to use any existing technique as discussed above for realizing variability in business logics while relying on our extension to trigger the dynamic switching among alternative behaviours.

Fig. 11 presents an example of this extended use of Web Service Context. This example simulates the business logic of the method “getQuote4ResidentialWithExtra”. The annotation “@Resource” (line 1) is used to denote that the variable context (line 2) is a Web Service Context resource which will be injected by the runtime engine. The resource context contains the initialized feature set for one service variant. According to this context resource, the first code block (line 7) is only executed if the feature “Accidental Damage” is enabled for the corresponding service variant. Similarly, the second and third code blocks (lines 10 and 13) are controlled by the features “Fusion Cover” and “Extended Third Party Liability” respectively. As such, even though the same code snippet is used across multiple service variants, each service variant exposes correct behavior corresponding to its feature configuration.

5 GENERATIVE MIDDLEWARE FOR AUTOMATED APPLICATION ENGINEERING

In this section we describe a generative middleware that supports the deployment of the configurable service implementation developed in the preceding section and the automated derivation of service variants during the application engineering process.

5.1 Overview

Fig. 12 presents the architecture of our generative middleware for provisioning customizable services. This generative middleware is built upon conventional service middleware and supports the following three functions specific to provisioning customizable Web services:

- **Validation of customization requests according to constraints defined by a feature model.**
- **Dynamic derivation and deployment of service variants.**
- **Lifecycle management of service variants.**

In order to support the two operation modes that were illustrated in Fig. 2, a customizable service exposes two types of endpoint, customization endpoint and variant endpoint. The customization endpoint is used for exchanging customization messages during the service customization mode. In contrast, variant endpoints are dynamically created for exchanging messages related to consuming particular service variants during the service consumption mode. In general, a new variant endpoint is dynamically created each time a new valid customization request is received so that the requesting customer is able to invoke service capabilities specific to his requirements (i.e., a particular feature configuration). The use of one variant endpoint for each particular feature configuration helps to separate one individual service variant from another with a different set of service capabilities.

5.2 Components of the Architecture

The function of each component in the generative middleware is as follow:

- **Customization frontend:** This is a Web service for performing service customization. The customization frontend is an intermediary between service consumers and the customization engine. Upon the arrival of a customization request, it delegates the validation of the requested feature configuration to the Feature model manager. It also queries the Runtime customization and management engine for the processing of valid requests. Any error during the processing of a customization request will be cascaded to this component for the communication to customers.
- **Feature model manager:** This component manages feature models and validates requested feature configurations against feature models.
- **Variant profile repository:** This component keeps track of deployed service variants. In particular, it maintains information about customers, feature configurations and variant endpoints.
- **Configurable service implementation:** This is runtime deployment of configurable artefacts produced during the domain engineering process.
- **Runtime customization and management engine (or Runtime engine):** This component cooperates with the Variant profile repository to decide whether a variant endpoint already exists or a new one needs to be created. In the latter case, it resolves variability in the configurable service implementation and produces the software artefacts that make up the service variant. In addition, the Runtime engine dynamically deploys the derived service variant as well as monitoring the operation and lifecycle of deployed service variants.
- **Service variant:** This is the runtime deployment of
5.3 Runtime Variability Resolution

This section discusses how the runtime engine resolves the variability in the configurable service implementation given a feature configuration. The two types of variability, as discussed in Section 4.3, are considered separately in two subsections.

5.3.1 Resolving Variability in Service Skeleton

The runtime engine uses reflection techniques [11] to resolve variability in service skeleton. Fig. 13 illustrates this operation using the Swinsure Insurance Web service. Firstly, the runtime engine constructs a runtime service model comprising service components, portType components and service operation components based on JAX-WS annotations (cf. the left hand side of Fig. 13). The runtime engine also examines the FeatureMapping annotations associated with components of the model. Based on FeatureMapping annotations, the runtime engine builds up the presence condition for each variant service capability.

The runtime engine then evaluates the constructed presence conditions against the feature configuration according to which the corresponding service variant is derived. Fig. 13 shows one example of such feature configurations around the arrow in the middle of the figure. As the status of all variant features is determined, each presence condition will be evaluated to TRUE or FALSE. If the presence condition is TRUE, the service capability is retained in the resulting service variant. Otherwise, the service capability is purged from the capability of the service variant. As the result, the runtime service model for a service variant is constructed.

For the feature configuration example in Fig. 13, the evaluation of the presence condition for the portType component Interface("purchasingPortType") results in its absence since the feature "Purchase" is disabled. Similarly, other components like Operation("getQuote4Residential") and Operation("getQuote4Business") are also removed from the original runtime service model. In summary, the use of FeatureMapping annotations and reflection techniques enables the runtime resolution of variability in the service skeleton.

5.3.2 Resolving Variability in Business Logic

The runtime engine supports variability resolution in business logic by initializing the service context before deploying a service variant. The service context contains information about what features are TRUE (i.e., selected) and what features are FALSE (i.e., disabled). This information enables the service variant to properly execute its business logic according to the corresponding feature configuration.

Service context initialization is done through the handler framework in JAX-WS. The handler framework is a flexible plug-in framework for message processing. In JAX-WS, handlers are organized into an ordered list called a handler chain. The handlers in a handler chain are invoked each time a message is sent or received. Example handlers are those for parsing XML messages, logging, or enforcing security policies.

In our framework, we extend JAX-WS with a new handler, called FeatureMappingHandler (Fig. 14). This handler puts into the message context of all incoming service consumption messages information about what features are selected/disabled. Also, the runtime engine adds this handler to the beginning of the handler chain for a service variant before deploying the service variant. As such, the deployed service variant is able to access to information about its feature configuration while executing its logic.

6 PROTOTYPE IMPLEMENTATION

6.1 Domain Engineering Tools

We have developed a set of Eclipse plugins (tools) for enabling the domain engineering process. These tools support feature modeling, service capability modeling as well as feature-capability mapping modeling as presented in Section 4.2.1, 4.2.2 and 4.2.3 respectively. The generation of service skeleton is supported by the WSDL2Java tool which is an implementation of JAX-WS in Apache CXF. For variability in service skeleton, we identify the presence condition associated with each variant service capability from the feature-capability mapping model. The FeatureMapping annotation type is then used to annotate the presence condition for the corresponding variant service capability in the generated SEI. For variability in

![Runtime representation of a service variant](image-url)
business logic, Web Service Context is utilized in service implementation to switch among alternative behaviours.

6.2 Generative Middleware

The prototype generative middleware is built as an extension of Apache CXF. The generative middleware relies on Apache CXF for all conventional Web service-related processing. It provides new functions related to service customization by adding the components (as outlined in Fig. 12) on top of Apache CXF. Each component of the generative middleware is implemented as a Java component performing the role defined in Section 5.2.

In particular, the component Customization frontend is implemented as a Web service which accepts a feature configuration and returns the URI from which the WSDL of a service variant can be retrieved. The component Feature model manager is implemented on top of the open source FAMA (FeAture Model Analyser) framework for automated analysis of feature models [12]. The component Variant profile repository is a Java component that interacts with a backend database to enable the registration and management of variant endpoints for deployed service variants.

The Runtime engine is another Java component that uses reflection techniques to perform dynamic variability resolution and derive service variants’ implementation as outlined in Section 5.4.1. Apache CXF uses interceptors to implement the handler framework defined by JAX-WS. Therefore, we define a new interceptor that plays the role of FeatureMappingHandler and adds the interceptor to the beginning of the inbound interceptor chain to facilitate the resolution of variability in business logic.

6.3 Client Customization Tool

We have extended our feature modeling tool to support service customization. The tool works as a Web service client with respect to the component Customization frontend. It allows customers to configure a feature model and use the resultant feature configuration to dynamically query a customizable service. This leads to the dynamic derivation and deployment of service variants. Fig. 15 presents the capture of such a transaction. The screenshot shows the feature configuration constructed at the customer’s side and the customization response returned by the Customization frontend of the Swinsure Insurance Web service.

Using the customization tool, we have confirmed the proper operation of the generative middleware:

- The middleware correctly processes valid customization requests and rejects invalid ones;
- Variability in both service skeleton and service implementation is properly resolved. In particular, service variants based on different feature configurations have different service capabilities. In addition, two service variants with different sets of extra covers produce different quotes even though they both expose the service operation “getQuote4ResidentialWithExtra()”;
- The lifecycles of the deployed service variants are properly managed. In particular, customizing the service with the same feature configuration results in the same variant endpoint. In addition, variant endpoints only exist during the period specified in the deployment policies.

7 Evaluation

In this section we present the evaluation of the generative middleware. In particular, we use a large-scale Amazon Web Services (AWS) case study to compare the traditional super-service approach and service customization approach.

7.1 AWS Case Study

The AWS is a cloud computing service that allows customers to obtain and manage virtual machines on Amazon’s cloud infrastructure [13]. The AWS WSDL defines a complete set of service operations for managing machine images, machine instances, subnets and configuring security. It has totally 149 service operations, a large number of data types, and spans 7253 lines of XML code. There is only one single WSDL for all customers regardless of their varying needs and demands. As such, the AWS is designed as a super-service.

We have redesigned the AWS as a customizable service using our service engineering process. The AWS feature model has 29 variant features and yields 75096 feature configurations. As we are unable to obtain AWS’s internal service implementation, we generated a dummy implementation for each service operation. The customizable AWS was successfully deployed on our generative middleware. Details of this engineering process can be found from the first author’s thesis [14].

The large number of possible service variants (75096) illustrates the fact that for this case study the multiple-variant approach is infeasible. The viable solution using traditional technologies is the super-service approach as exploited by AWS. To compare our service customization approach and the super-service approach, we implemented and deployed a super-service AWS on our generative middleware.

7.2 Experiment Setting

Among 75096 possible service variants, we have selected service variants from three categories for our performance study as follows:

- **Light-weight service variants**: These service variants require only small subsets of service capabilities
specified in the AWS WSDL. Fig. 16 presents the common denominator (or the base feature configuration) representing these service variants. Crossed (or filled) features are disabled (or selected) features for all service variants, while selectable features (i.e., ones with empty checkboxes) represent features which are selected for some service variants and disabled for others. For instance, all light-weight service variants do not require functions for managing subnets or external storages. As such, feature “Subnet management”, “External storage” and all of their children are disabled. 288 light-weight service variants have been selected for this experiment.

* Medium-weight service variants: These service variants require a moderate number of service capabilities offered by the AWS. In particular, all of them include functions for retrieving instance description, while some of them also include advanced features for managing external storage. In terms of the AWS feature model, feature “Instance description” and its children are selected, while feature “External storage” and its children are selectable. 315 medium-weight service variants have been selected for the experiment.

* Heavy-weight service variants: These service variants require almost all service operations provided by the AWS. As shown in Fig. 16, feature “Instance description”, “External storage” and their children have already been selected in the base feature configuration. 384 heavy-weight service variants have been selected for the experiment.

Feature configurations generated from the base feature configurations in Fig. 16 were used as customization requests to the generative middleware to dynamically derive service variants.

All experiments are performed in a Windows 7 64 bit PC with Intel Core i5 processor, 2.4 GHz and 4GB memory. During the experiments, the PC is kept in a clean condition with minimal running processes to guarantee that all experiments are as isolated as possible.

7.3 Experiment Result

7.3.1 Leanness of service interface

For each service variant dynamically deployed by the runtime engine, we have measured the number of service operations exposed on its service interface. Fig. 17 represents the relative measure with respect to that of the super-service AWS which has 149 service operations. As can be seen from the graph, medium-weight service variants require more service capabilities than light-weight service variants while heavy-weight service variants provide the largest number of service operations. The experiment result also shows that our generative middleware helps to shrink the service interface of service variants, simplifying the consumption of the AWS.

7.3.2 Memory footprint

With respect to the memory footprint of the deployed service variants, we have collected the following two metrics: i) the number of service implementation classes loaded in
the Java Virtual Machine (JVM) and ii) the size of memory consumption for provisioning a service variant (cf. Fig. 18).

These metrics are shown as the proportion of the corresponding measure for the super-service AWS. In general, service variants require fewer classes to be deployed at runtime. In addition, the provisioning of these service variants consumed less memory compared to the super-service. The memory consumption reduction and the reduction in the number of loaded classes are significant in the case of light-weight and medium-weight service variants. This is proportional to the number of service capabilities required by these service variants. In the case of heavy-weight service variants, the reductions are not significant as these service variants require a large number of service capabilities, accessing nearly all capabilities exposed by the AWS. Even in this case, the experiment result shows that memory footprint required by redundant service capabilities has been effectively purged by the generative middleware. In addition, as cloud services become a commodity, we believe that the number of customers requiring light-weight or medium-weight service variants will be dominant. Consequently, the significant reduction of the memory footprint for these categories is more pivotal.

7.3.3 Service provisioning time
The efficiency in more compacted service interfaces and smaller memory footprints for dynamically deployed service variants comes with the cost of longer service provisioning time. For super-service, the service provisioning time is only the deployment time. In contrast, for dynamically deployed service variants, the service provisioning time, as discussed in Section 5, is comprised of: i) feature reasoning time (i.e., time for reasoning the validity and pre-existing condition of a customization request), and ii) service variant derivation and deployment time (i.e., time for resolving variability in service implementation and deploying the derived artifacts). However, it is worth noting that this service provisioning time happens only once in the lifecycle of a service variant.

The provisioning time of the AWS super-service was 2.6s in our experiment. We observed that there is no difference in feature reasoning time for all service variants of the three categories. In particular, for all service variants, the feature reasoning time was about half of the provisioning time of the super-service. This is expected as this processing is similar across all service variants.

Fig. 19 shows the summary of the service variant derivation and deployment time compared to the provisioning time of the super-service. Due to the necessity of variability resolution, deriving and deploying light-weight service variants was 1.1-2.1 times longer than the super-service. For medium-weight and heavy-weight service variants, as the runtime service model gets bigger and more classes need to be loaded, the provisioning time increased. For the worst case scenario, the derivation and deployment time is 2.3 times longer than the super-service. In this case, the total provision time (including the feature reasoning time) is calculated as 2.3 * 2.6 + 1.3 = 7.28s. While this is much longer than the provisioning time for the super-service, it is negligible as this is only a one-off process during the much longer lifecycle of the dynamically deployed service variant.

8 Related Work
In this section, we discuss related work in three major categories: i) service variability modeling and service customization, ii) Software Product Line (SPL), and iii) feature-based customization of software systems.

Service variability modeling and service customization
There are only a few works that consider engineering techniques for developing and provisioning customizable Web services in the literature ([15], [16]). Stollberg et al. [15] define a set of meta-models for modeling variability in service interface models. Customers are able to use a variability model to customize the service interface according to their requirements. On the other hand, Liang et al. [16] extend the WS-Policy framework to incorporate customization policies. Customization policies describe how information in the service interface description can be altered (e.g., adding new elements or removing existing attributes from data types). Customization requests are built upon customization policies and are used by
service providers to enact changes to the service interface. The main focus of these works is to prune the service interface according to customer requirements. Variability management has not been properly considered, especially for variability in the service implementation and its runtime resolution. Also, service customization is performed at the technical level of service interface description. As discussed in the Introduction, this approach of service customization is more complex than our feature-based service customization approach.

A number of works have tackled the issues of modeling service variability (e.g., [17], [18]). Their main focus is on identifying variability in the service interface. Only Jiang et al. [19] take one step further to consider variability in UML class diagrams for service implementation. However, none of these works supports the comprehensive modeling and management of variability from the highest level of abstraction (i.e., the feature level) to the lowest level of abstraction (i.e., the service implementation level). Neither do they maintain the mapping of variability between different levels of abstraction. Therefore, their variability modeling techniques are not applicable to defining a service customization approach that minimizes the complexity of service customization. Also, their approaches are not able to support dynamic variability resolution for service variant derivation and deployment.

**Software Product Line (SPL)**

In conventional SPL, the binding of variation points typically occurs before the delivery of the software. Dynamic Software Product Line (DSPL) is an emerging field that aims to defer such binding until runtime in order to produce software that is “capable of adapting to fluctuations in user needs and evolving resource constraints” [20]. The concepts of DSPL have been applied to a number of application domains, such as ubiquitous computing [21] and service robotics [22]. Our framework can be considered as a DSPL approach. As far as we know, this is the first attempt that applies the concepts of DSPL to enabling customizable Web services. From the technical perspective, our contributions to DSPL research are the service engineering process and techniques, and the extension to the JAX-WS standard to realize variability at the service implementation level. Also we define the architecture of a generative middleware that enables dynamic variability binding and runtime provisioning of customizable Web services.

The mapping between feature models and software artifacts at lower levels of abstraction has been the focus of a number of works in SPL to align variability between different levels of abstraction (e.g., [23], [24]), to maintain the traceability of variability realization (e.g., [25]), and to enable automated derivation of a product (e.g., [22], [26]). Many of these works consider the mapping between feature models and architecture/design models. Our approach is similar to those works that support automated derivation of products in the way that we also capture the mapping at the implementation level. However, the key difference is that our solution is designed to work with Web service engineering. In particular, we build the runtime service model and utilize the mapping information to resolve variability in this model to dynamically derive service variants.

**Feature-based customization of software systems**

Feature models have been used in a number of frameworks for enabling customization (e.g., [27], [28], [29]). However, in most frameworks, feature models are used as the internal knowledge to reason about and configure software systems. The consumer profile approach [28], the service lines approach [29] and our work stand alone in the sense that we use feature models as a customer-facing software artifact to trigger customization. The consumer profile approach extends the WSRP (Web Services for Remote Portlets) protocol to enable the exchange of feature models and feature configurations between customers and providers. In contrast, we define the customization endpoint and a customization mode for this purpose so that such interaction can be performed in a service-oriented way. With respect to the service lines approach, our framework is different in two key aspects as different application domains are targeted. Firstly, we support the mapping between multiple features and a software artifact while the mapping in the service lines approach is one-to-one. Secondly, we allow the generation of service variants with different service interfaces while in the service lines approach, the service interfaces are required to be the same to enable the composition of workflow-based SaaS applications.

**9 Conclusion**

In this paper, we have presented a feature-based framework for developing and provisioning customizable Web services. Our framework is novel since it effectively addresses two major issues in delivering customizable Web services. They are the complexity in selecting the right subset of service capabilities for service variants and a systematic way for managing variability in service development and runtime provisioning. In addition to articulating the service engineering process, we have proposed a generative middleware that extends conventional service platforms for incorporating customizable services and enabling runtime service customization.

To facilitate the development and provisioning of customizable services, we have developed a set of tools for service engineering, a prototype generative middleware for service deployment and a customization tool for performing service customization. On top of this prototype system, we have successfully developed and deployed two case studies with different scales: the Swinsure Insurance case study and the AWS case study. The development of these case studies helped to evaluate and revise the service engineering process. In addition, the AWS case study was used to evaluate the efficiency achieved with the generative middleware. The experiment results show that for light-weight and medium-weight customer requirements, our framework helps to filter a large number of redundant service capabilities. Also, the framework enables the saving of runtime resource required by the deployment and provisioning of service variants as the result of the dynamic variability resolution.
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


Tuan Nguyen received the BEng degree in Information and Communication Engineering from the Faculty of Engineering, The University of Tokyo in 2006, the MSc degree in Frontier Informatics from the Graduate School of Frontier Sciences, The University of Tokyo in 2008, and the PhD degree in Software Engineering from the Faculty of Information and Communication Technology, Swinburne University of Technology, Melbourne, in 2012. He is currently a post-doctoral research fellow at the Swinburne University of Technology. His research interests include service computing, model-driven software engineering, Software Product Line and variability modeling, and cloud computing.

Alan Colman completed a Ph.D. in adaptive software architectures at Swinburne University of Technology in 2006. His current research focuses on customisation of service and context-aware systems, the application of control theory to autonomic software systems, and predictive performance modelling for services. He is currently leading a project on service customisation with Australia’s Cooperative Research Centre in Smart Services (Smart Services CRC). He has published more than 70 peer-reviewed papers in international journals and conference proceedings.

Jun Han received his B.Eng. and M.Eng. in Computer Science and Engineering from Beijing University of Science and Technology in 1992 and 1993 respectively, and his Ph.D. in Computer Science from the University of Queensland in 1992. From 1992 to 2003, he was with the University of Queensland and Monash University. Since 2003, he has been Professor of Software Engineering in the Faculty of Information and Communication Technologies at Swinburne University of Technology. He has also been a research leader with Australia’s Cooperative Research Centre in Smart Services (Smart Services CRC). Han and his associated CRCs have published more than 2000 peer-reviewed papers in international journals and conference proceedings. His research interests include adaptive and context-aware software systems, services engineering and management, software and system architectures, software security and performance, and system integration, evolution and interoperability.