

## VARIABILITY MODELING TO DEVELOP FLEXIBLE SERVICE-ORIENTED APPLICATIONS\*

Joonseok PARK<sup>1</sup> Mikyeong MOON<sup>2</sup> Keunhyuk YEOM<sup>1</sup>

<sup>1</sup>*Department of Computer Science and Engineering, Pusan National University, Busan, 609-735, Korea*  
*pjs50@pusan.ac.kr, yeom@pusan.ac.kr (✉)*

<sup>2</sup>*Division of Computer and Information Engineering, Dongseo University, Busan, 617-716, Korea*  
*mkmoon@dongseo.ac.kr*

### Abstract

To cope with requirement changes flexibly and rapidly, the existing component-based paradigm is being evolved into a service-oriented computing paradigm. The main characteristic of the service-oriented computing paradigm is that service-oriented applications are developed as loosely coupled services that reflect business concerns. This paradigm also promotes business agility, facilitating quick reactions to business changes. Therefore, to enhance and support the benefits of the service-oriented computing paradigm, we must consider how to improve flexibility and reusability during the development of service-oriented applications. We propose the variability modeling approach to specify and control the common and distinguishing characteristics of service-oriented applications. That is, the key concepts of product-line technology can be used to make service-oriented applications more flexible and reusable. This paper describes variability modeling at two levels; the composition level and the specification level. At the composition level, we describe the variability of composition and the flow of domain services that fulfill business processes. At the specification level, we present a domain service that is an abstract service with variability. The use of our systematic variability modeling approach can greatly increase the flexibility, applicability, and reusability of service-oriented applications.

**Keywords:** Variability, software product line, reuse, service oriented software, flexible service model

### 1. Introduction

The existing component-oriented paradigm (Kwon et al. 1999) focuses on information technology (IT) concepts. It has a limited ability to reflect requirement changes easily and allow

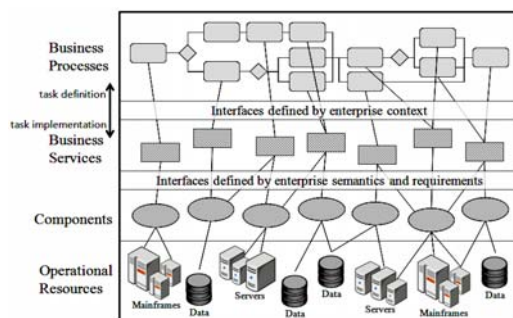
business flexibility. A service-oriented architecture (SOA) (Erl 2008) has been proposed to support the emerging service-oriented computing paradigm and hence reduce the gap between business and IT

---

\*This work was supported by the Grant of the Korean Ministry of Education, Science and Technology(The Regional Core Research Program/Institute of Logistics Information Technology) and by the National Research Foundation of Korea Grant funded by the Korean Government(MEST) (NRF-2010-20100328000)

concepts (Huhns et al. 2005, Bichler et al. 2006).

SOA is an architectural paradigm and discipline that may be used to build infrastructures that enable those with needs (consumers) and those with capabilities (providers) to interact via services across disparate domains of technology and ownership (Nickul et al. 2007). For most businesses, SOA offers considerable flexibility in aligning IT functions with business processes and goals (Arsanjani et al. 2007). In general, SOA can be considered as comprising of four layers, as shown in Figure 1 (Rosen 2006). The first layer is the business layer, which shows the business processes. The second layer is the service layer, which defines the services that realize the business processes. The third layer is the application layer. A service is realized as an IT component. The lowest layer illustrates the applications, packages, and databases that might be called upon by various components.



**Figure 1** Service-oriented architecture (Rosen 2006)

In SOA environments, service-oriented applications are developed as loosely coupled services. Therefore, the new service-oriented application can be developed by replacing loosely coupled services. Notionally and ideally, the SOA environments pursue structural flexibility, facilitating the development of new

service-oriented applications by replacing loosely coupled services. However, currently, difficult technical issues are encountered with regard to replacing loosely coupled services. Furthermore, we need to redesign and redevelop the services each time to develop service-oriented applications that reflect somewhat different functions in each application, but that performs similar functions in a domain. For example, consider that a previously developed order service has functions providing offline ordering, such as functions associated with a telephone or fax. When we require another order application that provides for online ordering, we should redevelop and recompose an order service that provides the online order function. In addition, many recently proposed approaches (Arsanjani 2004, Papazoglou et al. 2006, Mittal 2006) are not sufficient when dealing with service-oriented applications that provide similar functionality, and can be replaceable in a domain. In other words, these approaches are lacking when it comes to flexibility and reusability in application development. When the requirements in a system change, the function of a pre-existing service is added or a new service is developed. Therefore, to systematically develop service-oriented application families that provide similar functionality and provide better reusability and flexibility, we need an approach that interpret and control the common and distinguishing characteristics of services and service-oriented applications. One of the approaches that aim to extend and maximize flexibility and reusability is the variability concept, which has been proposed in software product line engineering (SPLE).

SPLE (Clements et al. 2002) is an approach used for developing software families using core assets. Its key concept is analyzing commonality (common features) and variability (distinguishing features) (Moon et al. 2005) in software development. This is called variability modeling (Clauß 2001, Sinnema et al. 2007); it helps in identifying and systematically developing common parts for reusability and is used in approaches involving product lines and system families. By using common and variable assets, various similar software families that are reusable and flexible can be developed in a short time. Therefore, we need to combine these commonality and variability concepts into service-oriented application development.

This paper focuses on variability modeling that can be used to support the development of service-oriented applications. By designing and developing domain services that have common and variable features, service reusability is increased through the reuse of existing common features and domain services to generate various services. In addition, service flexibility is achieved by selecting variable features, which enables the developers to respond more

promptly to a business change.

To extend flexibility and service reusability, we propose using variability modeling at both the composition level and the specification level. This variability modeling approach uses key concepts from product-line engineering, namely, commonality and variability. We propose a metamodel that describes the two-level variability modeling that is explicitly explained by commonality and variability. In addition, we illustrate our variability modeling approach via a case study in the supply-chain domain. We define a common variability model that can be reused as a core asset. The core assets for the development of service-oriented applications are the services and the loose coupling of these services. Therefore, by combining the product-line variability concept with these two types of core assets, we can construct new types of concrete service-oriented applications. Figure 2 shows how flexible service-oriented applications can be constructed from these assets. From this common model, we can generate loose couplings of services by selecting variable services, as shown on the right-hand side of Figure 2. In addition, each service that is

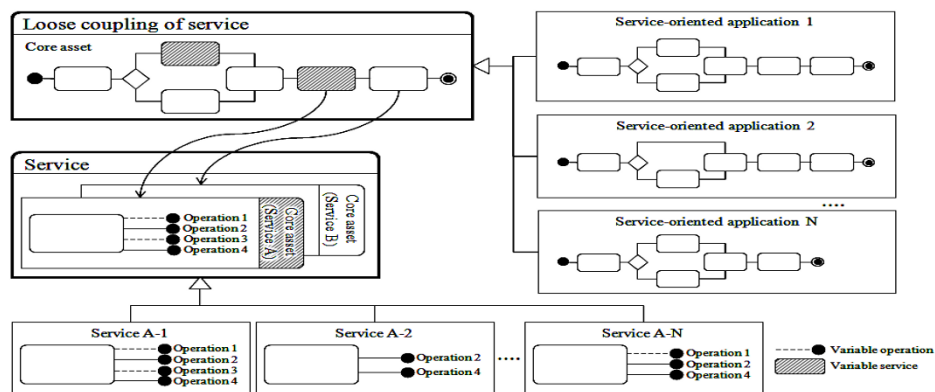


Figure 2 Development of service-oriented applications with flexibility based on product-line concepts

constructed in a loosely coupled manner has a variable operation. As shown at the bottom of Figure 2, we can generate different services by selecting variable operations. Therefore, by using a common variability model, we can realize a flexible service-oriented application.

It is difficult to develop service-oriented applications that provide similar functionality, but that change parts of a service. In other words identifying and modeling variability is complicated. Therefore, in this paper, we propose an approach that explicitly represents variability by analyzing it. By explicitly analyzing the predictable variable part of the variability modeling approach, we can design and develop services that reflect both the predictable variable part and the changeable part of the system, according to changing business requirements. By reusing these services as business requirements change, we promote business agility, and maximize flexibility through service replacement. In addition, we can increase productivity when developing service-oriented applications using this approach.

The rest of this paper is organized as follows. Section 2 introduces related works. Section 3 explains our metamodel for variability modeling. Specification-level and composition-level variability for the supply-chain domain are presented in detail in Section 4. Section 5 describes a case study and supporting tools for the proposed variability modeling. Our conclusions are given in Sections 6.

## 2. Related Works

Before explaining our variability modeling approach, we explain related works on SOA and

services, software product line engineering, and service modeling.

### 2.1 SOA and Services

SOA establishes an architectural model that aims to enhance the efficiency, agility, and productivity of enterprise software systems by positioning services as the primary focus (Erl 2008). In service-oriented computing, the basic element is the service.

Similar to the term “component”, there exist various definitions of a service. Bichler et al. (2006) defines a service in terms of being assigned its own distinct functional context and being comprised of a set of capabilities related to this context. Papazoglou et al. (2007) defines services as well-defined, self-contained modules that provide standard business functionality and are independent of the state or context of other services. Fremantle et al. (2002) defines services in terms of being described in a standard definition language, having a published interface, and communicating with each other to request execution of their operations, aiming to collectively support a common business task or process. From these perspectives, a service is an execution task that reflects more business aspects than IT concepts. In this paper, we define a domain service as an abstract and generalized service that has commonality and variability and that defines and generates the service layer’s service.

Next, we describe service level variability analysis (Chang et al. 2007, Segura et al. 2007). Chang et al. (2007) focus on service adaptation, classifying workflow, composition, interface, and logic variability. This approach focuses on the unit service. However, our model focuses on

the domain service, from which the model can generate unit service variants. In addition, Chang et al. (2007) use a simple tag representation for variability, whereas our model has more refined and specific model elements to represent variability. Segura et al. (2007) focus on web services and unit service invocation, classifying a binding time that divides design time and runtime, partner selection criteria, message exchanges, and protocols. However, their approach does not consider variability types at our level, and does not show how to represent variability. Our variability analysis approach views variability from two aspects, namely, a specification level and a composition level. We also present explicit methods and elements to represent variability.

## 2.2 Software Product Line Engineering

A product line is a set of software intensive systems sharing a common, managed set of features, which satisfy the specific needs of a particular market segment or mission, and which are developed from a common set of core assets in a prescribed manner (Clement et al. 2002). A product line comprises two steps, namely, domain engineering and application engineering. Reusable core assets are developed in the domain engineering step. In the application engineering step, core assets are customized and new applications are developed according to application-specific requirements. In this approach, the key concept is variability, which refers to assumptions about how members of a family may differ from each other (Weiss et al. 1999). Figure 3 shows a metamodel of the fundamental concepts of product-line architecture.

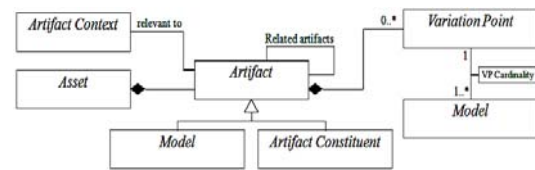


Figure 3 Variability metamodel for a product-line artifact

An asset provides a collection of artifacts. An artifact is a work product that can be created, stored, and manipulated. The artifact contains elements that specify the model in a specific domain. An artifact context helps explain the meaning of the elements in the artifact. A variation point (VP) is the point at which variant binding occurs. Variants refer to particular instances of realized variability. VP Cardinality denotes the number of variants that can be applied to the variation points. In this paper, we include the product line concept in our proposed variability model.

The Process Family Engineering in Service-Oriented Applications (PESOA) group has proposed the variability mechanism for modeling process family architectures (Bayer et al. 2005). Their product family architecture contains information about the realization of variability in contrast to other variability mechanisms. This approach focuses on business processes. However, our approach performs analysis at a different abstract level that is based on a domain service. In addition, we refine variability at the domain service level and present an explicit model. In particular, with respect to the product line architecture, an important task is the analysis of the domain and the identification of the commonality and variability of the domain.

The Feature-Oriented Reuse Method (FORM) (Kang et al. 1998) was developed as an

extension of the Feature-Oriented Domain Analysis (FODA) method (Kang et al. 1990). The main characteristic of FORM is its four-layer decomposition, which describes different points of view for product development. However, this does not explicitly address the variations in the reference architectures, and leads to complexity when many variants must be represented. Keepence et al. (1999) represented variation using patterns associated with discriminates. A discriminate has three types, namely, single, multiple, and optional, and is closely related to the division of feature properties into mandatory, optional, and alternative. However, it does not emphasize the characteristics of variation at the design level.

Gomma (2004) explained the product line design phase in connection with features. He endeavored to describe design models with explicit variations in structural and dynamic views. However, all possible variants appear at the same level in these models, and consequently, they are complex, even for simple case studies.

### 2.3 Service Modeling

A number of researchers have investigated the service composition modeling approach using business process execution language for web services (BPEL, WS-BPEL or BPEL4WS) (Alvis et al. 2007, Zhai et al. 2008, Saab et al. 2009, Michlmayr et al. 2010), web service choreography interface (WSCI) (Arkin et al. 2002), and Petri nets (Hamadi et al. 2003, Valero et al. 2009). BPEL is an XML-based specification language for specifying business processes that are exclusively based on web services. In addition, BPEL has been proposed by leading players in industry (BEA, IBM, and

Microsoft), and has quickly become a standard (Alvis et al. 2007), and supports process-oriented service-composition. WSCI specifies the overall collaboration between web service providers and web service users by describing message exchanges between those involved. The proposed BPEL and WSCI approaches are based on XML. An XML-based representation has the advantage of being a universal representation for data exchange, but it can be difficult to understand and write for non-XML experts (Skogan et al. 2004). In addition, this modeling approach only considers the single composition case of a service-oriented application. In contrast, our approach considers multiple composition cases, and shows replaceable relationships of services using explicit types of variability. Furthermore, with respect to composition, we extended UML activity diagrams, which are a standard in graphical modeling languages. Therefore, a user can easily and efficiently model the composition of a system.

A Petri net is a directed, connected, and bipartite graph. Service composition is modeled as a Petri net by assigning transitions to methods and places to states. This approach focuses on formalism and does not consider interpreting and controlling the common and distinguishing characteristics of services. In addition, there is a graph-based approach (Liang et al. 2006, Liu et al. 2006, Hashemian et al. 2005, Lang et al. 2005) dealing with the service composition problem. Liang et al. (2006) propose four classes of service specification graphs: single-source single-destination (SSSD), multiple-source single-destination (MSSD), multiple-source multiple-destination (MSMD),



and directed acyclic graphs (DAGs). This approach focuses on finding low-cost service composition solutions. However, our approach focuses on designing and developing a flexible service. Liu et al. (2006) propose that for composite service discovery, web services be represented as graphs. In their approach, each node denotes a web service, and each arc denotes the relationship between web services. In addition, its focus is using a web service graph for service discovery, whereas our model focuses on variability modeling of service composition. Hashemian et al. (2005) use a dependency graph in which a node represents the inputs and outputs of web services and edges represent the associated web services. This dependency graph is used for searching among web services to find those whose composition provides a specific behavior. These proposed approaches use a graph for searching and discovering service-composition aspects. In addition, Lang et al. (2005) propose using general AND/OR graph to represent all possible input-output dependencies among the web services registered in some selected service categories. It presents AND/OR graph search algorithms used for composite service discovery. In addition, it expresses operation nodes as AND nodes and data entity nodes as OR nodes. This approach focuses on searching for, and discovering of a service.

Tut et al. (2002) propose using patterns in service composition. They propose a payment mechanism pattern, and the two generic patterns: a project pattern and a maintain pattern. The payment mechanism pattern shows that different mechanisms can be carried out for billing and payments. The project pattern describes a

systematic method of making and following a plan. The maintain pattern describes how to assess a situation and make a decision to repair or improve the situation. This approach shows that patterns can be applied to some composition aspects, such as payment. Zirpins et al. (2004) introduce service interaction patterns that specify generic process characteristics. It proposes generic mechanisms that allow us to represent relationships, or coordination policies within the abstract service composition logic, which are the interaction patterns and their concrete coordination choices, or coordination idioms. These pattern-based approaches can represent static-type replacement, which means identifying a matching composition case, and then having the user represent it according to this fixed pattern. In addition, it is difficult to patternize composition cases. However, our approach offers more dynamic replacement, in which replaceable services can be bound to variation points. The main difference is that our approach deals with the predictable variable aspects of developing service oriented applications with similar functionality in a domain.

In this paper, we use a UML activity diagram and expand it to model and design service-composition aspects. By considering the variability in service composition, we can generate different cases of service-composition, so as to deal with a change in requirements. Therefore, our approach focuses more on the method of developing and designing a reusable composite service than do other approaches.

### 3. Domain Service MetaModeling

In this section, we present a two-level

variability metamodel for a domain service to support the development of flexible service-oriented applications. For this, we extend the variability metamodel for a product-line artifact shown in Figure 3 to obtain the two-level variability metamodel shown in Figure 4, which driven by the metamodel of a product-line artifact shown in Figure 3. Our metamodel for a domain service is expressed as a specialized version of that shown in Figure 3. Each concrete concept has a stereotype, which indicates the corresponding concept in the metamodel for a product-line artifact. As shown in Figure 4, our variability metamodel for a domain service is specified at two levels, namely, the composition level and the specification level.

The composition level shows the domain-service flow and the composition variability. A domain composition model is represented in terms of domain services and their flow, and the composition relationship variability information describes how domain services are loosely coupled and composed to

realize flexible service-oriented applications. Therefore, by selecting composition-level variability, we can generate new versions of one or more variants of domain-service compositions.

At the specification level, we focus on those domain services that comprise the domain-service composition model. It has the domain-service specification model, which describes domain service information, such as variation points and variants. This level specifies details of a domain service, such as the properties, operation, and messages. By selecting variability in domain services, we can generate new versions of one or more variants of the domain services.

Therefore, our proposed metamodel is sufficiently flexible to handle changing requirements because it has predictable differences as variability. Similar to a Lego block, at the composition level, we consider many different flow compositions for domain

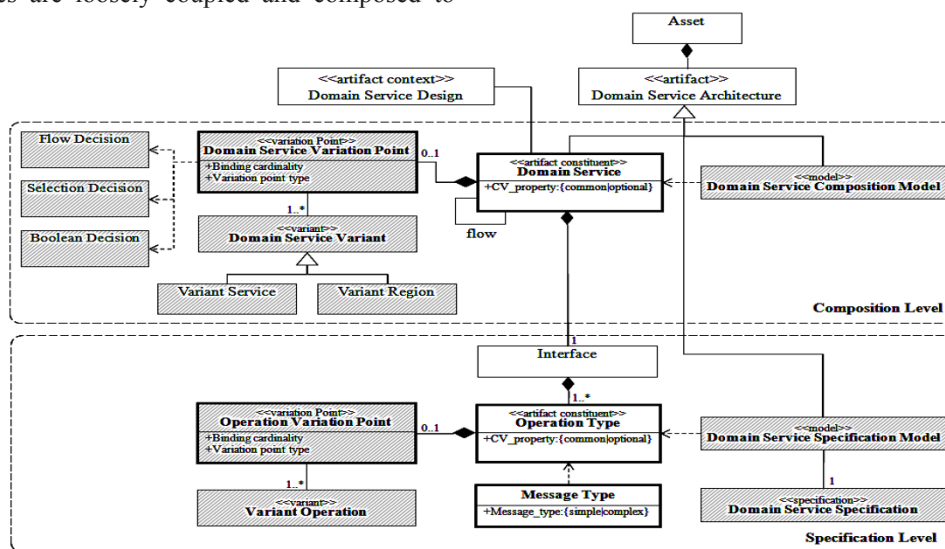


Figure 4 Two-level variability metamodel for a domain service



services. For example, the purchasing application consists of optional and common domain services: *authentication*, *buying*, *payment*, and *delivery*. According to the requirements *authentication* may be an optional service, and the flows between *payment* and *delivery* may be indeterminate. Therefore, purchasing applications may employ different orders. For example, one type of the purchasing application may follow the flow *authentication*, *buying*, *payment*, and *delivery*, whereas another may follow *buying*, *delivery* and *payment*, but not *authentication*. Thus, to represent these changeable statuses, our model has a domain service composition model with a domain-service variation point, and domain-service variants, and shows the flow of these domain services. Furthermore, at the specification level, a domain service itself has a predictable difference. For example, the interface for the authentication service includes operations that to *authenticate by password* or *certificate of authentication*. According to requirements, the interface may consist of just one default operation called *authenticate by password*. Furthermore, it may also have two types of operations. To represent these options, our model has a domain service specification model, which contains the variability information of a domain service, the operation type, and the message type.

As shown in Figure 4, the basic elements for constructing a variability model are the domain service, the interface, the operation type, and the message type.

*Domain service*: an abstraction and generalization of services that uses commonality and variability to define and generate the service

layer's service. It expresses its variability in terms of the CV\_property, which can be common or optional. It is a reusable and executable business activity, and it can be invoked or used by other users for business task execution.

*Interface*: an access and connection point for outside elements. A domain service has one interface.

*Operation type*: an abstraction and generalization of a domain service operation that can be invoked or provided.

*Message type*: the input or output data type for an operation.

In addition, the shaded parts of Figure 4 show the expanded elements for construing a variability model.

*Domain-service composition model*: one common model that can be reused as a core asset for developing loose coupling of domain services. It shows domain services and their flow, and the composition relationships.

*Domain-service specification model*: one common model that can be reused as a core asset for developing a domain service. It shows the basic information of the domain services, with variability such as operation type and message type.

*Domain-service specification*: one specification that explains a domain service.

*Domain-service variation point*: there are four types of variation points, namely, addition/omission variability, alternative variants, flow decision and flow condition.

*Operation type variation point*: this implies that the operation type of a domain service is realized as mandatory or optional for selective candidate operations.

*Boolean decision*: a decision on whether or not to use domain service variant.

*Selection decision*: a decision on whether to select a single or multiple domain service variants.

*Flow decision*: a decision that selects between sequence, parallel, or condition relationships between domain services.

Variability modeling that is based on this proposed metamodel is explained further in Section 4.

## 4. Variability Modeling

Variability modeling is explained at both the composition and the specification level. We describe variability modeling in detail using supply chain management applications, i.e., the supply chain domain. A supply chain management application manages and controls the overall distribution from warehouse to markets, retail outlets, and convenience stores. The system provides services, such as the warehousing of goods, taking goods out of the warehouse, returning goods, and ordering goods.

### 4.1 Variability Modeling at the Composition Level

A domain-service composition model is defined as one common model that can be reused as a core asset for developing loosely coupled domain services. To explain our domain-service composition model, we use the Unified Modeling Language (UML) activity diagram notation, and expand it to describe the variability concept proposed by Moon et al. (2008).

Table 1 shows the expanded UML elements and their meaning. A domain action can be

common or optional. A common action is one that should be included in most applications. An optional action is one that may only be included in specific applications.

As shown in Figure 5, a domain-service composition model includes domain services and the relationship between them. That is, it comprises domain actions and their relationship. Domain actions are mapped to each domain-service operation type. Therefore, each domain action is expressed as a domain-service name and a domain-service operation type. In the case of the variability of alternative variants, the domain action is expressed as an abstracted action name. We explain addition/omission variability, variability of alternative variants, flow condition variability, and variability of flow decisions further in Section 4.2.

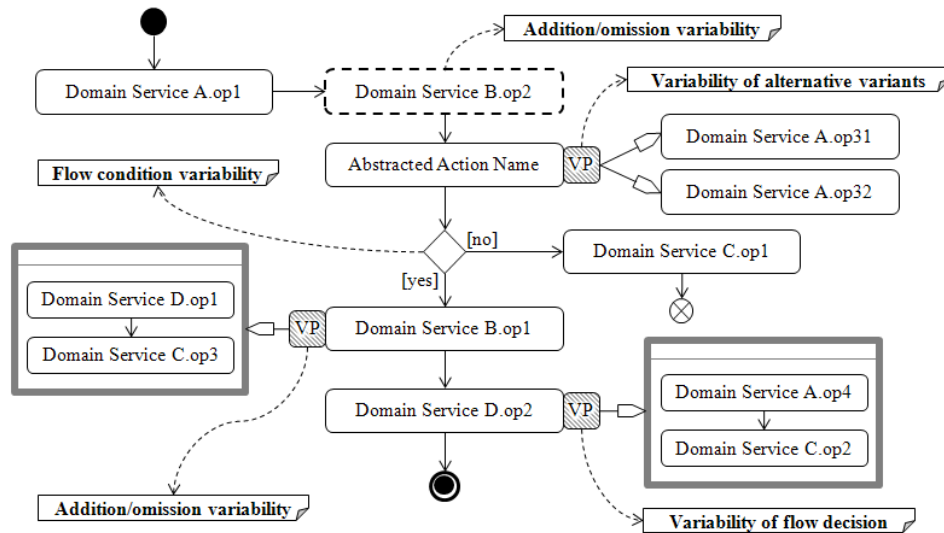
### 4.2 Variation Point Type at the Composition Level

To represent composition-level variability, we use and refine the variability concept proposed by Moon et al. (2008). Here, the composition-level variability type is subdivided into four subtypes, namely, addition/omission variability, variability of alternative variants, variability of flow decision, and flow condition variability. We represent these types of variability in the domain-service composition model. The different types of variability at the composition level are as follows:

*Addition/Omission variability*: a domain action or grouped domain action, is added to, or omitted from the domain-service flow. Figure 6 shows an order business flow. In this case, the common domain action is that of receiving orders from stores, retail outlets, and markets.

**Table 1** Notation for domain service composition model

Element	Description	Notation
Optional domain action	Represented by a dashed ellipse	
Domain action variation point	A box labeled VP is attached to a domain action at the variation point	
Variation point binding	Binds a variation point and a variation	
Variant region	A rectangle surrounds a grouped domain action	



**Figure 5** Domain-service composition model based on proposed notation

However, according to the requirements, an optional offline order may also be included. Therefore,  $\text{Order}(\text{domain service name}).\text{opOfflineOrder}(\text{domain service operation type})$  represents the optional domain action. There is also a group addition/omission for a domain action. For example, if orders are approved for registered users, registration will be an expanded business task. We model the expanded domain action group by using a variant region. In this manner, registration domain actions can be added or omitted.

*Variability of alternative variants:* this indicates that one or more domain actions have alternative or replacement relationships that can be generalized into abstract domain actions. In

Figure 7, one replacement means that only one replaceable domain action variant can be selected. In the supply chain domain, order information checking is a common domain action. In this situation, checking per time period and checking per day are alternative domain actions. Therefore, the abstract domain action is the accept order and  $\text{Order.opPerTime}$  and  $\text{Order.opPerDay}$  are the replaceable domain actions. There are also one-to-many replaceable relationships. A variation point with cardinality 1..2 means that one or two variants are available for selection.

*Variability of flow decision:* this indicates that the related domain action flow cannot yet be decided. This type of variability decision is

delayed as long as necessary. Undecided domain actions are grouped in a variant region, as shown in Figure 8. When the order is completed, we are not able to decide on certain domain actions, such as send order invoice, update order records, and review order records, because the flow relationship depends on the option that is most preferable. Therefore, these domain actions remain open. These flow relationships can be decided as sequential, parallel, conditional, or compound (a composition of sequential, parallel, and conditional flow elements).

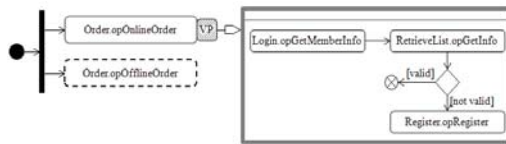


Figure 6 Example of addition/omission variability

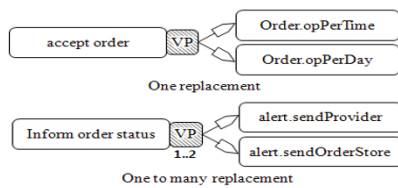


Figure 7 Examples of variability of alternative variants

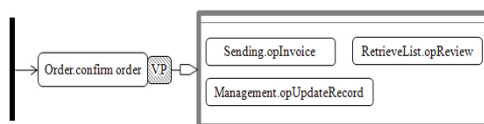


Figure 8 Example of flow decision variability

*Flow condition variability:* this indicates that the domain action has a conditional branch. The guard condition for a decision node generates another path of domain action flow. Therefore, we define this variability as flow condition variability. Figure 9 shows the conditional acceptance of a decided provider list. If

acceptance occurs, the request order domain action is invoked.

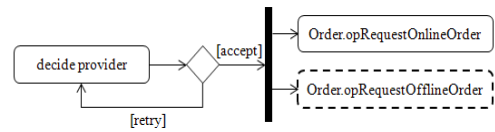


Figure 9 Example of flow condition variability

### 4.3 Variability Modeling at the Specification Level

A domain-service specification model is defined as one common model that can be reused as a core asset for developing the domain service. Based on the proposed variability metamodel, we define a notation for representing the domain-service specification model, as listed in Table 2.

Using the notation proposed in Table 2, the domain-service specification model is represented as shown in Figure 10. Common/optional variability and operation-type variability of a domain service are explained further in Section 4.4.

### 4.4 Variation Point Type at the Specification Level

To represent specification-level variability, we divide the variability type into three subtypes, namely, common/optional variability, operation type variability, and message type variability. These types of variability are represented in the domain-service specification. Variability at the specification level is classified and explained as follows:

*Domain-service common/optional variability:* this indicates that variability occurs in the domain service itself. If the domain service is a

specific service, and the domain service is to be included in most service-oriented applications in the domain, it is a common (mandatory) domain service. Otherwise, it is an optional domain service.

*Operation type variability:* this is expressed in terms of the CV property, which can be common, denoted by <<c>>, or optional, denoted by <<p>>. The variation point comprises a generalization of two or more candidate operations. This variability means that a domain service’s operation type is realized as mandatory or optional for selective candidate operations. There are four possible cardinalities: [1, 0..1, 0..N, 1..N]. Cardinality [1] means that this operation type is specified as one mandatory candidate operation. Cardinality [0..1] means that this operation type can be specified as one optional candidate operation. This cardinality type candidate operation cannot have a variation point. Cardinality [0..N] means that this operation type can be specified as two or more alternative candidate optional operations. Cardinality [1..N] means that this operation type can be specified as two or more alternative mandatory operations. For the [0..N] or [1..N] relation, one default candidate operation can be specified. We denote this operation as <<default>>. Each candidate operation is

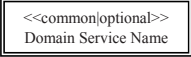


documented as operation\_name(in:MessageType, out:MessageType). Operation type (in:) means it is an input message, and (out:) means it is an output message.

*Message type variability:* this involves the input and output messages of an operation type. Input and output messages can differ in either number or type. In the case of variability in number, we generalize this message as being of a complex type. Message type variability comprises the message property, the complex type name, and the message structure. The message property describes this message as an input or output. The complex type name describes the complex type. The message structure is documented as <<variability info>>element name:type. For cases that include options, we describe the <<variability info>> part as <<optional>>.

Table 3 shows a domain-service specification template that involves variability information. Each domain service is described using the proposed template. In this table, operation type variability is represented in the operation type part, and message type variability is represented in the message type.

Based on the proposed domain-service specification template, Table 4 shows an example of order domain service specifications in the supply chain domain.

**Table 2** Notation for domain-service specification model

Element	Description	Notation
Domain Service	Represented as a rectangle containing common/optional property	
Interface	Represented by a lollipop shape	
Operation Type	Represented by a rounded rectangle. It shows the common/optional property, denoted by <<c p>>, and the binding cardinality, represented by [vpCardinality] operationName with input(in)/output(out) messages.	

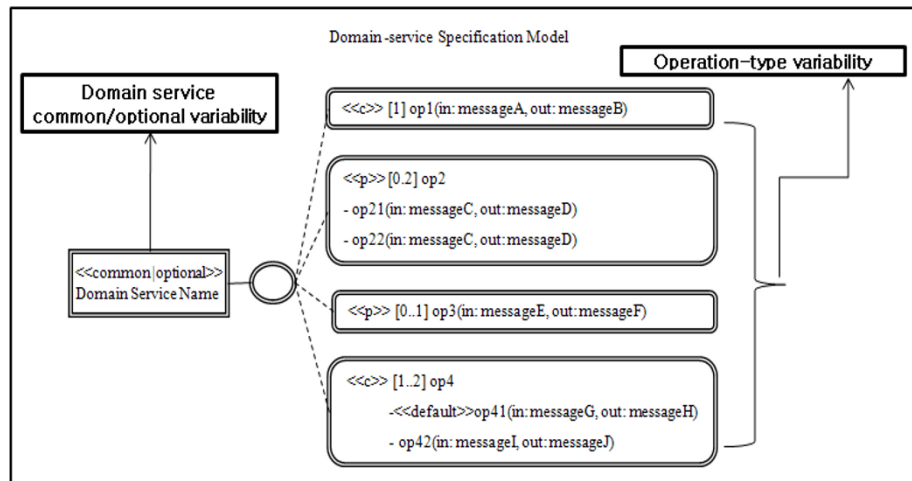


Figure 10 Domain-service specification model based on the proposed notation

Table 3 Domain-service specification template

Domain-service name	Describe domain-service name Describe domain-service information	Property	Describe domain service's CV_property (common optional)
Domain-service interface	Describe domain-service interface		
<b>Operation Type</b>			
Operation Property	Variation Point	Cardinality	Operation
Describe Operation type's CV_property – denote <<c>> in case of common, <<o>> in case of optional	Describe Generalized Operation. In case of cardinality [1],[0..1], Operation is described	1 0..1 1..N 0..N	Operation1(in:MessageType, out:MessageType) Operation2(in:MessageType, out:MessageType) Operation3(in:MessageType, out:MessageType) :
<b>Message Type</b>			
Message Property	Complex Type Name	Message Structure	
Describe message as input or output – in: in case of input message, out: in case of output message	Describe complex type name	Element name : type <<optional>>Element name : type :	

Operation type variability is shown in the order domain-service example in Table 4. Here, the order operation can be online [opOnlineOrder] or offline [opOfflineOrder]. Online ordering can be a common operation with binding cardinality [1], and offline ordering is selective, i.e., with binding cardinality [0..1]. One common or optional candidate operation's

variation point is not assigned. In addition, the Accept order variation point is a common operation in cases in which there are alternative relations, such as checking per time period [opPerTime] or checking per day [opPerDay]. Note that we describe the default operation using the <<default>> notation. Message type variability is also shown in the example in Table



4. The operation input message, ProductInfo, for opOnlineOrder and opOfflineOrder contains mandatory data [orderer, productid, quantity] and the optional data [message, payment method]. Note that we describe the optional data using the <<optional>> notation.

### 5. CASE Study and Supporting Tool

To investigate the effectiveness of our variability model, we conducted a case study involving a supply chain system. In addition, we developed a supporting tool to support our two-level variability modeling. Figure 11 shows the scenario for using the supporting tool. As shown in this figure, a user can model the

proposed variability model at the composition level and the specification level. In addition, the user can decide the variability of the model. Therefore, this case study shows how to explicitly represent variability. This means that our variability modeling approach represent and explicitly analyzes the predictable variable part, with respect to designing and developing similar service-oriented applications.

We organized two development groups in terms of similar education backgrounds, programming skills, and knowledge of the service-oriented computing environment. Figure 12 shows the domain services for the supply chain system. For example, Order is a domain

**Table 4** Order domain service

Domain-service name	Order This domain order service encapsulates the order business task.	Property	Common
Domain-service interface	Order Interface		
Operation Type			
Operation Property	Variation Point	Cardinality	Operation
<<c>>	opOnlineOrder(in:productInfo,out:onlineorderInfo)	1	-
<<p>>	opOfflineOrder(in:productInfo,out:offlineorderInfo)	0..1	-
<<c>>	Accept order	1..2	<<default>> opPerTime(in:time) opPerDay(in:date)
<<p>>	Verify order	0..3	opConfirmMsg(in:orderInfo, out:confirmstatus) opConfirmSMSmsg(in:orderInfo, out:confirmstatus) opConfirmMailmsg(in:orderInfo, out:confirmstatus)
..	..	..	..
Message Type			
Message Property	Complex Type Name	Message Structure	
in	productInfo	orderer:string productid:string quantity:int <<optional>>message:string <<optional>>payment method:string	
out	confirmstatus	confirm:bool rserved:bool cancel:bool	
..	..	..	

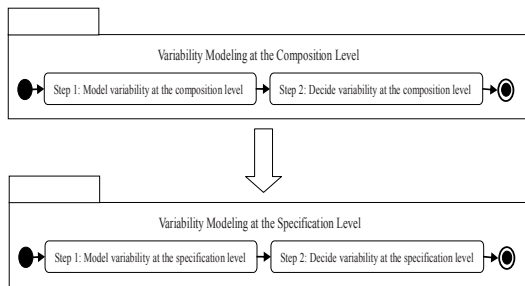


Figure 11 Scenario for using the supporting tool

Domain Service (Number of candidate operation) - Descriptions
<input type="checkbox"/> Order (10) – the ordering of goods
<input checked="" type="checkbox"/> Management (8) – provider decision information
<input checked="" type="checkbox"/> RetrieveList (10) – the search for data related to the order, warehouse, and stock
<input checked="" type="checkbox"/> Register (3) – registration of customers
<input checked="" type="checkbox"/> Sending (4) – the sending of related document and data
<input checked="" type="checkbox"/> WarehousingService (5) – the warehousing of goods
<input checked="" type="checkbox"/> DeliveryService (4) – delivering of goods from the warehouse
<input type="checkbox"/> WarehouseCRUD (7) – the registration, modification, deletion and searching of warehouse information
<input checked="" type="checkbox"/> StoreCRUD (7) – the registration, modification, deletion and searching of stock information
<input checked="" type="checkbox"/> Login (3) – authentication of users
<input checked="" type="checkbox"/> WarehouseStock (10) – stocktaking for the warehouse
<input checked="" type="checkbox"/> ProductCRUD (7) – the registration, modification, deletion and searching of product information
<input checked="" type="checkbox"/> SellService (5) – the selling of the product
<input checked="" type="checkbox"/> ReturningService (5) – the returning of goods
<input checked="" type="checkbox"/> StorestockAdmin (3) – stocktaking for the stock
<input checked="" type="checkbox"/> Alert (4) – notification of event message

Figure 12 Domain services

service name, (10) means that the number of the order domain service’s candidate operation is 10, and description means the order domain service handles the ordering of goods.

Figure 13 shows a snapshot of the variability-modeling tool used with a service-composition model of the ordering operation of supply-chain management. It reflects the variability at the composition level, as described previously. Specifically, it shows an explicit representation of variability types at the composition level, expressed using the VP and variability notation, for example, optionalAction, variationPoint, variantsRegion, and VPBinding.

### Step 1: Model the variability at the composition level

As shown in Figure 13, the right-hand

column shows the modeling notation, and the bottom pane defines the properties of the composition model. By dragging and dropping the modeling notation, the user can model the composition model in the center pane. In addition, the left-hand pane shows the object generated by the composition model. Therefore, Figure 13 shows the composition model that enables us to produce an ordering service-oriented application. The application has similar functionality within a domain, as mentioned in the core asset shown in Figure 2, which implies a loose coupling of services, with variability.

### Step 2: Decide the variability at the composition level

The composition model for the variability of the ordering application is decided by the supporting tool. Figure 14 shows a snapshot depicting the task of deciding the variability at the composition level. The left-hand pane of the figure shows the variability at the composition level. The right-hand pane shows the decision result, and the bottom pane shows, the properties of the variability information.

A variability decision is made by selecting the list of variabilities. As shown on the right-hand side of the window, a user can make a Boolean decision. That is, the user can decide whether to use a variant in the case of addition/omission variability. Furthermore, in the case of variability of alternative variants, the user can decide between a single selection and multiple selections based on the binding cardinality. After deciding on the variability, the window displays the generation case of the composition model, which shows the composition flow of a domain service. Thus, as

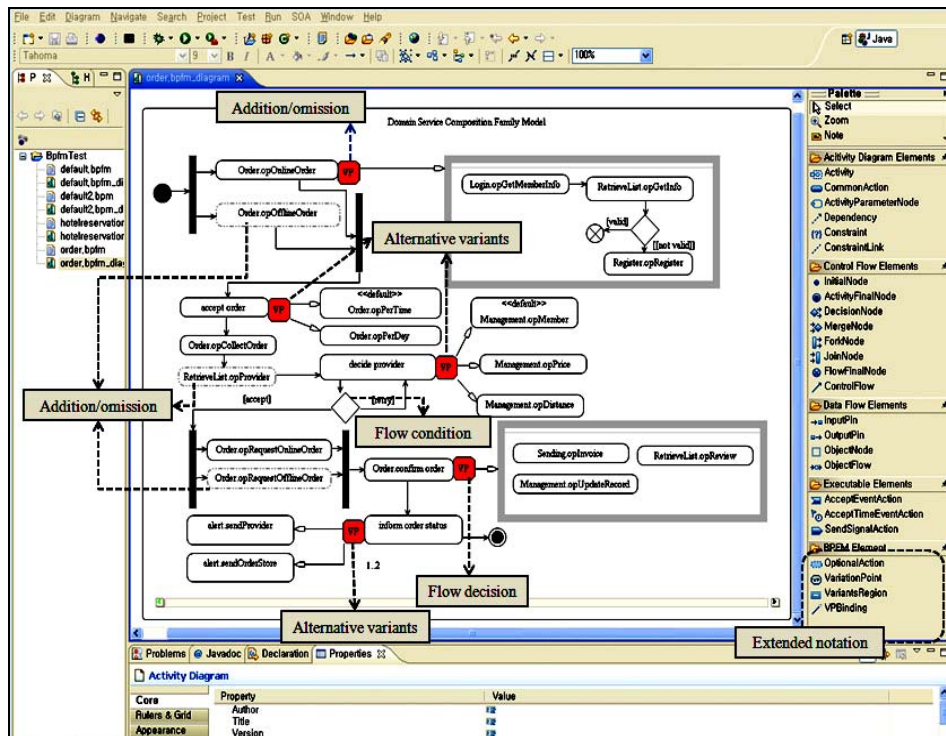


Figure 13 Composition model for an ordering application

business requirements change, we can relatively easily generate different service-oriented applications which providing an ordering function by selecting the correct variable service (refer to Figure 2). This case study shows how variability modeling at the composition level supports and guides the generation of composition models by deciding on variability.

### Step 3: Model the variability at the specification level

Figure 15 shows a snapshot of the tool used for variability modeling at the specification level. Similar to the tool used for modeling composition variability, the user can drag and drop the modeling notation in the left-hand pane and model the variability at the specification level. Figure 15 shows the order services that have variable operations in the composition

model shown in Figure 13.

As shown in Figure 15, the order domain-service has common and optional operation types. Its common operation types are `opOnlineOrder`, `opCollectOrder`, `opRequestOrder`, `opRequestOnlineOrder`, `opDisplay order` and a variant point named `Accept order`, which has variant operations `opPerTime` and `opPerDay`. In addition, its optional operation types are `opOfflineOrder`, `opRequestOfflineOrder`, `opDisplayOrder`, `opWishlistOrder`, `opRearrangeOrder` and a variant point named `Verify order`, which has variant operations `opConfirmMsg`, `opConfirmSMSmsg`, and `opConfirmMailmsg`. This domain service shows how to represent variability types at the specification level, including types such as `<<common>>`, `<<c>>`,

and <<p>>, with binding cardinalities. Therefore, it shows variable services that enable us to generate new services in a domain, as mentioned in the core asset shown in Figure 2 named Service A, which produced types named service A-1, service A-2, and service A-N.

**Step4: Decide the variability at the specification level**

Figure 16 shows that variant services can be generated by using reusable domain services, as explained in Figure 2. The variant service named order is an example of a common selection operation type. It comprises the common operations opOnlineOrder, opCollectOrder, opRequestOnlineOrder and opDisplayOrder. The variant operation opPerTime is also selected. Therefore, it shows how to reuse a domain service to generate new services.

Group A did not use variability model while Group B used the order domain-service composition variability model. In the first

development application (Number of Applications 1), Group A developed their applications more quickly than Group B. This was because Group B had to spend time analyzing and constructing a variability model. However, in the later order supply chain applications (Number of Applications 2 and 3), Group B’s development time was much shorter than that of Group A. Once the variability model had been developed, new service applications derived from the variability model can be developed easily and quickly. This is because Group B could reuse many pre-developed domain services, thereby reducing the development time for the similar applications.

After completing the development, groups agreed that, in similar service-oriented application families, our variability model is very helpful. This is because it contains systematic and well-organized variability

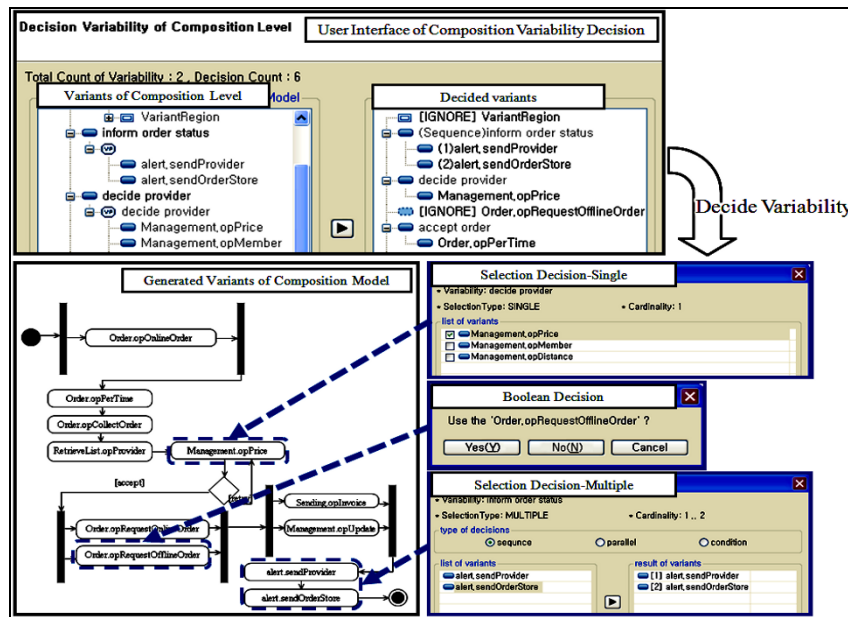


Figure 14 Snapshot depicting the task of deciding composition level variability

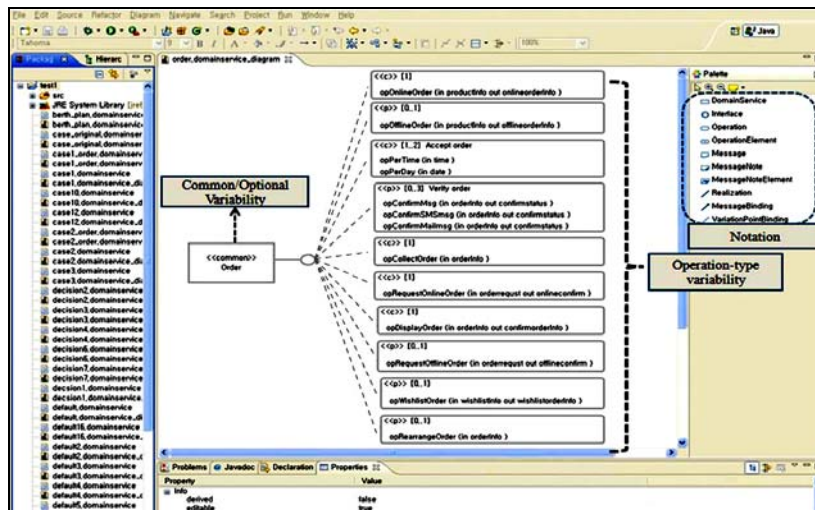


Figure 15 Specification model of the order domain service

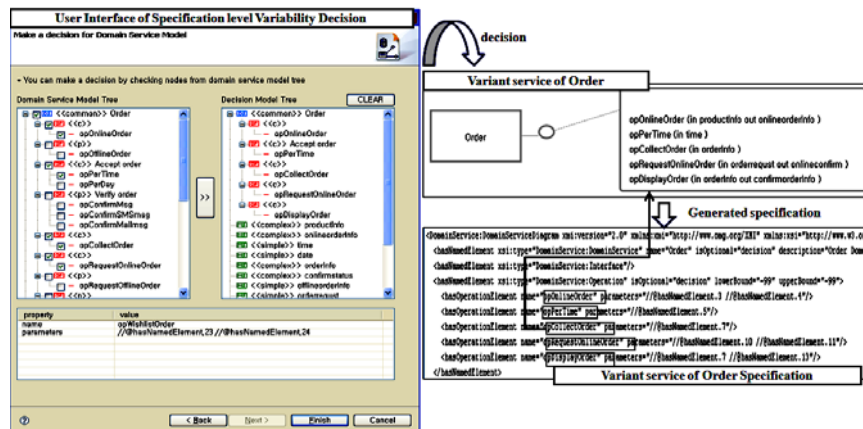


Figure 16 Order variant service generation based on order domain service

information, and indicates how the system can be constructed. Table 5 shows various variant generation cases of compositions and services that can be generated using the proposed case study. From this table, we recognize that our variability model can enhance the flexibility, applicability, and reusability of service-oriented applications. From common, well-defined variability models, we can generate various variants by deciding on the variability at the model level alone. This reduces the time

required to design and develop variant applications in a domain. Therefore, our variability modeling can guide, support, and be applied to the design and development of flexible service-oriented applications systematically and efficiently.

## 6. Conclusion

We have proposed a variability modeling approach to supporting the development of flexible service-oriented applications. First, we

Table 5 Variant Case

Composition level	Composition model – Figure 13	Case of variant composition
Addition/omission	Order.opOfflineOrder Order.opRequestOfflineOrder RetrieveList1.opProvider	-none: ${}_3C_0$ -one of them: ${}_3C_1$ -two of them: ${}_3C_2$ -three of them: ${}_3C_3$ Total: 8 type
Alternative	Order.opOnlineOrder-VP	-Use or Ignore Total: 2 type
	Accept order -Order.opPerTime -Order.opPerDay	-one of them: ${}_2C_1$ -two of them: ${}_2C_2$ Total: 3type
	Decide provider -Management.opMemeber -Management.opPrice -Management.opDistance	-one of them: ${}_3C_1$ -two of them: ${}_3C_2$ -three of them: ${}_3C_3$ Total: 7type
	Inform order status -alert.sendProvider -alert.sendOrderStore	-one of them: ${}_2C_1$ -two of them: ${}_2C_2$ Total: 3type
Specification level	Specification model – Figure 15	Case of variant service
Operation type	[0..1] opOfflineOrder opRequestOfflineOrder opWishlistOrder opRearrangeOrder	-none: ${}_4C_0$ -one of them: ${}_4C_1$ -two of them: ${}_4C_2$ -three of them: ${}_4C_3$ -four of them: ${}_4C_4$ Total: 15type
	[0..3] Verify order opConfirmMsg opConfirmSMSmsg opConfirmMailmsg	-none: ${}_3C_0$ -one of them: ${}_3C_1$ -two of them: ${}_3C_2$ -three of them: ${}_3C_3$ Total: 8type
	[1..2]Accept order opPerTime opPerDay	-one of them: ${}_2C_1$ -two of them: ${}_2C_2$ Total: 3type

define a metamodel for variability modeling, which incorporates the concept of two-level variability modeling. Second, we systematically and explicitly classify the variability types and present an approach to realize the specification and representation of this variability modeling. Within composition-level variability modeling, we identify addition/omission variability, variability of alternative variants, variability of flow decisions, and flow condition variability. In addition, we represent specification-level variability modeling in terms of domain-service common/optional variability, operation type

variability, and message type variability. Third, we develop a supporting tool based on the proposed variability modeling. Using this variability modeling, we develop several types of supply-chain service-oriented applications. By using the proposed variability model, we can develop flexible service-oriented applications. This can enhance both efficiency and productivity in service-oriented software design and development.

We are now studying the methods and approaches to realize this domain service as an implementation-level service component. As



part of future work, we intent to design an explicit method for deploying and searching a domain service. In addition, these studies on the variability modeling approach, domain service realization, and deploying and searching will be extended to and integrated with process environments for flexible development of service oriented applications.

### Acknowledgment

The authors would like to thank the anonymous reviewers for their valuable comments.

### References

- [1] Alves, A., Arkin, A., Askary, S., Barreto, C., Bloch, B., Curbera, F., Ford, M., Goland, Y., Guizar, A., Kartha, N., Liu, C.K., Khalaf, R., König, D., Marin, M., Mehta, V., Thatte, S., Rijn, D.V.D., Yendluri, P. & Yiu, A. (2007). Web services business process execution language, version 2.0. Available via DIALOG. <http://docs.oasis-open.org/wsbpel/2.0/OS/wsbpel-v2.0-OS.html>. Cited April 11, 2007
- [2] Arsanjani, A. (2004). Service-oriented modeling and architecture: how to identify, specify and realize services for your SOA. IBM Developer Works. Available via DIALOG. <http://www.ibm.com/developerworks/library/ws-soa-design1/>. Cited November 9, 2004
- [3] Arkin, A., Askary, S., Fordin, S., Jekeli, W., Kawaguchi, K., Orchard, D., Pogliani, S., Riemer, K., Struble, S., Takacs-Nagy, P., Trickovic, I. & Zimek, S. (2002). Web Service Choreography Interface (WSCI) 1.0. Available via DIALOG. <http://www.w3.org/TR/wsci/>. Cited August 8, 2002
- [4] Arsanjani, A., Zhang, L.J., Ellis, M., Allam, A. & Channabasavaiah, K. (2007). S3: a service oriented reference architecture. *IT Professional*, 9 (3): 10-17
- [5] Bayer, J., Buhl, W., Giese, C., Lehner, T., Ocampo, A., Puhmann, F., Richter, E., Schnieders, A., Weiland, J., & Weske, M. (2005). Process family engineering-modeling variant-rich processes. PESOA-Report No. 18/2005, September 1, 2005
- [6] Bichler, M. & Lin, K.J. (2006). Service-oriented computing. *Computer*, 39 (3): 99-101
- [7] Chang, S., La, H. & Kim, S. (2007). A comprehensive approach to service adaptation. In: *International Conference on Service Oriented Computing and Applications*, 191-198, Newport Beach, California, USA, June 19–20, 2007, IEEE Computer Society
- [8] Clauß, M. (2001). Generic modeling using UML extension for variability. In: *Proceedings of OOPSLA Workshop on Domain Specific Visual Languages*, 11-18, Tampa Bay, Florida, USA, October 14–18, 2001
- [9] Clements, P. & Northrop, L. (2002). *Software Product Lines Practices and Patterns*. Addison-Wesley
- [10] Erl, T. (2008). *SOA Principles of Service Design*. Prentice-Hall
- [11] Fremantle, P., Weerawarana, S. & Khalaf, R. (2002). Enterprise services. *Communications of the ACM*, 45 (10): 77-82
- [12] Gomma, H. (2004). *Designing Software Product Lines with UML: From Use Cased to Pattern-Based Software Architectures*.

- Addison-Wesley
- [13] Hamadi, R. & Benatallah, B. (2003). A Petri net-based model for web service composition. In: Australasian Database Conference, 17: 191-200, Adelaide, Australia, February, 2003, Australian Computer Society
- [14] Kang, K., Cohen, S., Hess, J., Novak, W. & Peterson, S. (1990). Feature-Oriented Domain Analysis (FODA) feasibility study. Software Engineering Institute, Carnegie Mellon University technical report CMU/SEI-90-TR-21, November 1990
- [15] Hashemian, S.V. & Mavaddat, F. (2005). A graph-based approach to web services composition. In: 2005 Symposium on Applications and the Internet, 183-189, Trento, Italy, January 31–February 4, 2005, IEEE Computer Society
- [16] Kang, K., Kim, S., Lee, J., Kim, K., Shing, E. & Huh, M. (1998). Form: a reuse oriented reuse method with domain specific reference architectures. *Annals of Software Engineering*, 5 (1): 143-168
- [17] Keepence, B. & Mannion, M. (1999). Using patterns to model variability in product families. *IEEE Software*, 16 (4): 102-108
- [18] Huhns, M. & Singh, M.P. (2005). Service-oriented computing: key concepts and principles. *Internet Computing*, 9 (1): 75-81
- [19] Kwon, O., Yoon, S. & Shin, G. (1999). Component-based development environment: an integrated model of object-oriented techniques and other technologies. In: International Workshop on Component-Based Software Engineering, 47-53, May 17–18, 1999
- [20] Lang, Q.A. & Su, S.Y.W. (2005). AND/OR graph and search algorithm for discovering composite web services. *International Journal of Web Services Research*, 2 (4): 46-64
- [21] Liang, J. & Nahrstedt, K. (2006). Service composition for generic service graphs. *International Journal on Multimedia Systems*, 11 (6): 568-581, Springer Berlin/Heidelberg
- [22] Liu, J. & Chao, L. (2006). Web services as a graph and its application for service discovery. In: International Conference on Grid and Cooperative Computing, 293-300, Changsha, Hunan, China, October 21–23, 2006, IEEE Computer Society
- [23] Michlmayr, A., Rosenberg, F., Leitner, P. & Dustdar, S. (2010). Web services compositions modelling and choreographies analysis. *International Journal of Web Services Research*, 7 (2): 87-110
- [24] Mittal, K. (2006). Service oriented unified process. Available via DIALOG. <http://www.kunalmittal.com/html/soup.html>
- [25] Moon, M., Hong, M. & Yeom, K. (2008). Two-level variability analysis for business process with reusability and extensibility. In: International Conference on Computer Software and Applications, 263-270, Turku, Finland, July 28–August 1, 2008, IEEE Computer Society
- [26] Moon, M., Yeom, K. & Chae, H.S. (2005). An approach to developing domain requirements as a core asset based on commonality and variability in a product line. *IEEE Transactions on Software Engineering*, 31 (7): 551-569
- [27] Nickul, D., Reitman, L., Ward, J. & Wilber, J. (2007). Service Oriented Architecture (SOA) and specialized messaging patterns.

- Adobe Systems Incorporated White Paper. Available via DIALOG. [http://www.adobe.com/enterprise/pdfs/Services\\_Oriented\\_Architecture\\_from\\_Adobe.pdf](http://www.adobe.com/enterprise/pdfs/Services_Oriented_Architecture_from_Adobe.pdf). Cited, December 7, 2007
- [28] Papazoglou, M.P. & Van Den Heuvel, W.J. (2006). Service-oriented design and development methodology. *International Journal of Web Engineering and Technology*, 2 (4): 412-442
- [29] Papazoglou, M.P. & Van Den Heuvel, W.J. (2007). Service-oriented architectures: approaches, technologies and research issues. *International Journal on Very Large Data Bases*, 16 (3): 389-415, Springer Berlin/Heidelberg
- [30] Rosen, M. (2006). BPM and SOA – where does one end and the other begin? *BPTrends Columns& Article*. Available via DIALOG. <http://www.bptrends.com>. Cited January, 2006
- [31] Saab, C.B., Coulibaly, D., Haddad, S., Melliti, T., Moreauz, P. & Rampacek, S. (2009). An integrated framework for web services orchestration. *International Journal of Web Services Research*, 6 (4): 1-29
- [32] Segura, S., Benavide, D., Ruiz-Cortes, A. & Trinidad, P. (2007). A taxonomy of variability in web services flows. In: *Service Oriented Architectures and Product Lines – What Is the Connection?* (SOAPL-07), Kyoto, Japan, September 10, 2007
- [33] Sinnema, M. & Deelstra, S. (2007). Classifying variability modeling techniques. *Information and Software Technology*, 49 (7): 717-739
- [34] Skogan, D., Grønmo, R. & Solheim, I. (2004). Web service composition in UML. In: *International Enterprise Distributed Object Computing Conference*, 47-57, Monterey, California, USA, September 20–24, 2004, IEEE Computer Society
- [35] Tut, M.T. & Edmond, D. (2002). The use of patterns in service composition, web services, E-business, and the semantic web. *LNCS2512:28-40*
- [36] Valero, V., Cambroner, M.E., Diaz, G. & Macia, H. (2009). A Petri net approach for the design and analysis of web services choreographies. *The Journal of Logic and Algebraic Programming*, 78: 359-380
- [37] Weiss, D.M. & Lai, C.T.R. (1999). *Software Product-Line Engineering: A Family Based Software Development Process*. Addison-Wesley
- [38] Zhai, Y., Su, H. & Zhan, S. (2008). A reflective framework to improve the adaptability of BPEL-based web service composition. In: *International Conference on Services Computing*, 1: 343-350, Honolulu, Hawaii, USA, July 8–11, 2008, IEEE Computer Society
- [39] Zirpins, C., Lamersdorf, W. & Baier, T. (2004). Flexible coordination of service interaction patterns. In: *International Conference on Service oriented Computing*, 49-56, New York, USA, November 15–18, 2004, ACM Press
- Joonseok Park** received the bachelor's degrees in computer engineering from the Pukyong National University in 1999 and the MS degree in Computer engineering from the Pusan National University in 2002. Currently, he is a doctoral candidate in the Computer Engineering Department at the Pusan National University. His research interests include service oriented computing, ubiquitous computing, software

architecture, and product-line engineering.

**Mikyeong Moon** is an associate professor of computer and information engineering at the Dongseo University in Busan, KOREA. She received the BS degree in 1990 and the MS degree in 1992, both in computer science from the Ewha Womans University, Seoul, Korea, and the PhD degree in computer science and engineering from Pusan National University in 2005. Her current research interests include software reuse, product line engineering, requirement engineering, and RFID solutions. She is a member of the Korea Information Science Society.

**Keunhyuk Yeom** received the bachelor's degree

in computer science and statistics from Seoul National University in 1985 and the MS and PhD degrees in computer and information science and engineering from the University of Florida in 1992 and 1995, respectively. He is professor of computer science and engineering at the Pusan National University in Busan, Korea. From 1985 to 1990, he was with LG Electronics Research Institute. His research interests include software product lines, component-based software development, software reuse, self-adaptive software, situation-aware framework for RFID/USN Solutions, etc. He is a member of the ACM, the IEEE Computer Society, and the Korea Information Science Society.