Modeling and Managing the Variability of Web Service-based Systems

Chang-ai Sun [a,*], Rowan Rossing [b], Marco Sinnema [b], Pavel Bulanov [b], Marco Aiello [b]

[a] *Department of Computer Science and Technology, University of Science and Technology Beijing, 30 Xueyuan Road, Haidian District, 100083 Beijing, China*

[b] *Department of Mathematics and Computer Science, University of Groningen, P.O. Box 800, 9700 AV Groningen, The Netherlands*

Abstract

Web service-based systems are built orchestrating loosely coupled, standardized, and internetworked programs. If on the one hand, Web services address the interoperability issues of modern information systems, on the other hand, they enable the development of software systems on the basis of reuse, greatly limiting the necessity for reimplementation. Techniques and methodologies to gain the maximum from this emerging computing paradigm are in great need. In particular, a way to explicitly model and manage variability would greatly facilitate the creation and customization of Web service-based systems. By variability we mean the ability of a software system to be extended, changed, customized or configured for use in a specific context.

We present a framework and related tool suite for modeling and managing the variability of Web service-based systems for design and run-time, respectively. It is an extension of the COVAMOF framework for the variability management of software product families which was developed at the University of Groningen. Among the novelties and advantages of the approach are the full modeling of variability via UML diagrams, the run-time support, and the low involvement of the user. All of which leads to a great deal of automation in the management of all kinds of variability.

Key words: Service Engineering, Variability modeling, Variability management, Web services

* Corresponding author.
Email addresses: casun@ustb.edu.cn (Chang-ai Sun), rowan.rossing@itsround.nl (Rowan Rossing), mail@msinnema.nl (Marco Sinnema), p.bulanov@rug.nl (Pavel Bulanov), aiellom@cs.rug.nl (Marco

1 Introduction

Information systems today are not computational islands, but rather systems that need to communicate and interoperate over the Internet or corporate intranets. Supermarkets automatically order new products when stocks run low. On-line loan providers communicate with banks and loan registers. All these computer systems are different and there is no uniform way of accessing them, which complicates communication. Consider for example an on-line travel agency. If one wants to purchase a vacation package, the travel agency has to poll multiple companies to get the prices on airline tickets, hotels and rental cars. Each of these companies likely uses different, incompatible applications for pricing and reservations, making interaction more difficult (Curbera et al., 2002). Web services aim to solve the interoperability problem by providing a standardized way of exchanging data between these information systems. They use basic Web protocols for communication and are based on open XML standards, making them platform independent and developer friendly. Systems can be composed that are largely or entirely built on Web services. These systems are known as service-oriented systems, service-centric systems, or Web service-based systems (Curbera et al., 2002; Peltz, 2003).

Variability is the ability of a software system or artifact to be extended, changed, customized, or configured for use in a specific context (Sinnema et al., 2006a). Two important concepts related to variability are variation points and variants. Variation points are locations in the design or implementation at which variation will occur, and variants are the alternatives that can be selected at those variation points (Bachmann and Bass, 2001). Consider again the example of the on-line travel agency. Due to a dynamic network environment, the Web service of a particular airline can become unavailable. In that case, the Web service of a different airline that offers the same flight can be used. This kind of variability can be captured in a variation point for selecting a particular airline Web service. The variants in this case are the different Web services that can be selected.

COVAMOF [1] is a variability management framework, developed at the University of Groningen, to handle the issues in variability management relevant for the software industry (Deelstra et al., 2005; Sinnema et al., 2006a, 2004, 2006b). It offers facilities to model the variability in a software system over multiple layers of abstraction. The COVAMOF framework is designed specifically for use with software product families. The idea behind software product families is intra-organizational reuse through the explicitly planned exploitation of similarities between related products (Linden, 2002). Individual prod-

Aiello).

[1] `http://www.covamof.com/`

ucts are derived from a shared set of reusable components. The COVAMOF framework helps developers in deriving these individual products by providing an associated tool suite, called COVAMOF-VS, which is an add-in for Microsoft Visual Studio .NET (Microsoft Visual Studio .NET Web site, 2007). COVAMOF has already been validated to be very useful in industry (Deelstra et al., 2005).

It is important to consider variability management in Web service-based systems such as the system for the on-line travel agency. The dynamic execution environment of Web services makes it possible to change such systems at run-time; in fact Web services can be replaced or can be reconfigured to adapt to different circumstances. Explicit variability management in Web service-based systems provides the following advantages (Koning et al., 2009):

- It helps in meeting the Quality of Service. When a currently configured service performs inadequately, it can be replaced by a better performing one, or parameters can be changed to achieve better performance.
- It can enhance the availability of the system. When a service becomes unavailable, a backup service with the same functionality can be used as a replacement.
- It can be used to optimize the quality attributes, by changing the configuration of the system.
- It allows for run-time flexibility. Rebinding of services can be performed at run-time, and potentially *automatically* when needed.

Variability modeling for Web service-based systems differentiates from the one for traditional product families. The main difference lies in that the former has to provide more flexible run-time support due to the Service Oriented Architecture (SOA), while the latter focuses more on compile-time support. We now want to leverage the potential of the COVAMOF framework for variability management in Web service-based systems. This is a challenge, because the COVAMOF framework is geared towards software product families and therefore does not yet focus on run-time reconfiguration.

In (Koning et al., 2009), we took a first step towards the application of the COVAMOF framework to Web service-based systems. The Business Process Execution Language (BPEL) (Business Process Execution Language (BPEL) Web site, 2007) was extended to support variability. BPEL is an XML-based programming language that can be used to describe the interaction between Web services at the message level; in this way it also describes their composition. The newly developed language, called VxBPEL, has extra XML elements to support variation points and variants in a BPEL process. We used the COVAMOF framework to view the variability in VxBPEL processes. However, the approach of only using VxBPEL is not fully compatible with the COVAMOF framework, because not all of its variability concepts are sup-

ported. Also, with VxBPEL variability is still only modeled in the implementation layer, and not in higher layers of abstraction.

Thus, we focus on modeling variability also at the *architectural* level. Architectural modeling is important in Web service-based systems for the same reason it is important in software product families: it helps in understanding the composition of the system. Also, to make full use of the COVAMOF framework, we need to describe Web service-based systems at multiple layers of abstraction. These considerations are generalized in the following question: *"How can one model variability in the* architecture *of Web service-based systems, and can this variability be managed at run-time?"*

To answer these questions, we have designed and developed a profile for the Unified Modeling Language (UML) (Unified Modeling Language (UML) Web site, 2007) for modeling variability in Web service-based systems at the architectural level. This UML profile is conceptually compatible with the COVAMOF framework. We have also extended the COVAMOF-VS tool suite, to allow it to view and configure the variability in a Web service-based system. Furthermore, to manage the variability in Web service-based systems at run-time, we have proposed a variability management process that requires only minimal involvement from the end-user. This management process is driven from the COVAMOF-VS tool suite, and makes use of our approach for architectural variability modeling.

Incidentally, we remark that multiple views of the architecture may become inconsistent while making variability choices. We believe that ensuring the consistency of such multiple views of the architecture should be left to the software engineer. In other words, it is the responsibility of the software engineer to model the variability consistently over the different views of the architecture, or to employ a modeling tool that can detect inconsistencies. In the proposed approach, we use the UML tool (ArgoUML Web site, 2007), which provides some abilities of checking the inconsistency over different views.

In summary, this work includes the following contributions:

- A general extension to UML for modeling variability in UML diagrams.
- Full architectural modeling of variability of Web service-based systems via UML diagrams.
- A full application of the COVAMOF framework to Web service-based systems. This is a major change from product families.
- A management process driven by the COVAMOF-VS tool suite to automate the management of variability in Web service-based systems at run-time, with low involvement of the user.

The remainder of the paper is organized as follows. In Section 2, we provide three examples that are amenable to the techniques we propose here. In Sec-

tion 3, we describe the underlying concepts and techniques of our method. In Section 4, we define the UML profile we designed to model variability in the architecture of Web service-based systems. In Section 5, we describe how to use our UML profile to model the different types of variability that can occur in Web service-based systems. In Section 6, we propose a variability management process for managing variability in Web service-based systems at run-time. In Section 7, we describe our extensions to the COVAMOF-VS tool suite. In Section 8, we provide an overview of related work. The conclusion is reported in Section 9.

2 Examples

There are many examples of software products whose realization can benefit from variability modeling and management. Here we report three examples. First, a supply chain. A classical example for which we go into details and we use throughout the paper to exemplify concepts related to the proposed methodology. Second, we give an example of a controlled environment: that of customization of laws in local governmental bodies. Third, the case of adapting the same domotic software to many different houses by modeling variations.

2.1 Supply chain

As an example of a Web service-based system we use the application proposed in (Chapman et al., 2003) by the Web services Interoperability Organization (WS-I) (Web Services Interoperability Organization (WS-I) Web site, 2007), which is extended in (Baresi et al., 2003). The sample application describes a Supply Chain Management System (SCMS). It has an extensive architecture, consisting of multiple views, and the architectural diagrams are in UML.

Supply chain management is the process of planning, implementing, and controlling the operations of the supply chain with the purpose to satisfy customer requirements as efficiently as possible. The SCMS consists of consumer services, retailer services, warehouse services, shipping services, and manufacturer services. The consumer Web service can be a vendor Web site where consumers can order goods. There may be multiple retailer services, multiple warehouse services, multiple shipper services, and multiple manufacturer services. So, there are plenty of opportunities for variability.

The SCMS can be described at three layers of abstraction, i.e., the feature layer, the architectural layer, and the implementation layer. At the implementation layer the actual implementations of the Web services exist, but also

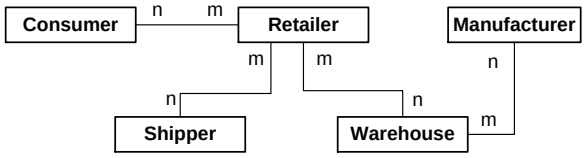

Fig. 1. Composition view of the SCMS.

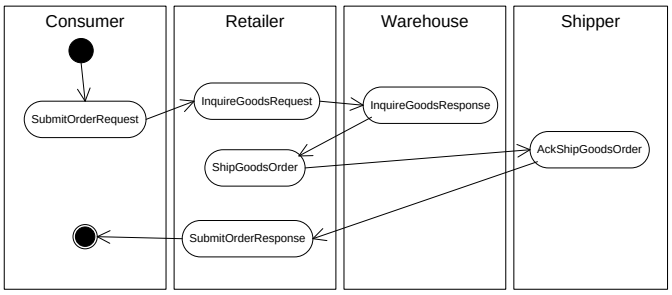

Fig. 2. Business process view of the SCMS.

WSDL files describing the interfaces of the services, and other files needed to deploy and run the system. The feature layer describes the high-level features and requirements of the system. However, our focus is primarily at the architectural layer, which describes the architecture of the system using multiple different views.

For the purpose of this paper, we have simplified the architecture of the SCMS to one UML diagram per architectural view. The architecture consists of the following views:

- The *composition view* models the composition of the Web service-based system. It uses *UML class diagrams* to model the Web services and their interconnecting relationships. Figure 1 shows the composition view of the SCMS architecture.
- The *business process view* models the processes executed by the Web service-based system. The individual processes are modeled through *UML activity diagrams*. Figure 2 shows the business process view of the SCMS architecture.
- The *use case scenario view* models possible use case scenarios in the Web service-based system. These exchanges of messages between Web services are modeled through *UML sequence diagrams*. Figure 3 shows the use case scenario view of the SCMS architecture.
- The *deployment view* details the distribution of Web services over the network. *UML deployment diagrams* are used for this. Figure 4 shows the deployment view of the SCMS architecture.

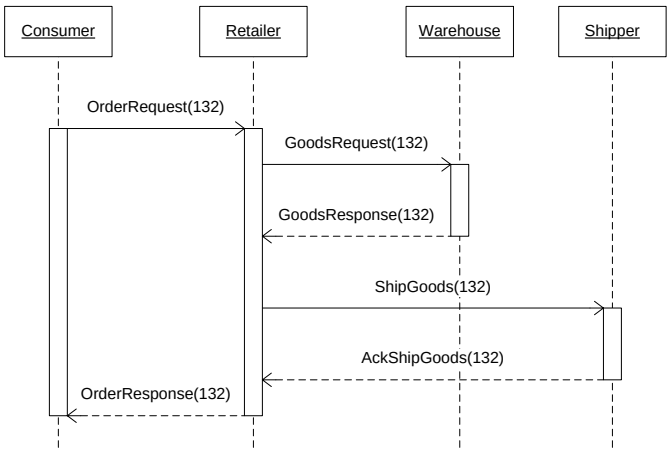

Fig. 3. Use case scenario view of the SCMS.

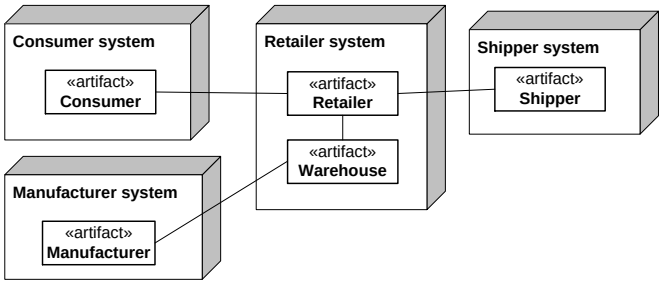

Fig. 4. Deployment view of the SCMS.

2.2 Local eGovernment

Most laws governing local bodies, such as municipalities, are defined by the central government and have an impact on the business processes and information systems of the local entities. If the laws can be formalized as formal process with variability taking into account the business and technical differences of the various municipalities, one can have great advantages from reuse. Consider the Dutch case where there are 441 municipalities and a regulation such as the WMO law [2] that mandates, for instance, the rules for providing publicly subsidized wheel chairs to citizens by the municipalities.

Now there are two roads to manage the translation from law to "instance of giving out a wheel chair in municipality X." One way is to give the interpretation document to all municipalities and let each one of them implement the law autonomously, as it is done today. The other way is to provide a formalized and generic process including variability describing the law and let the municipalities customize it according to their organizational structure and

[2] Wet maatschappelijke ondersteuning, Social Support Act approved in 2007 in the Netherlands.

their ICT infrastructure.

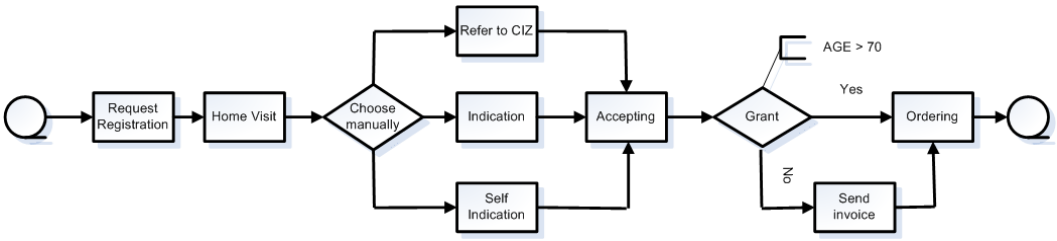

Fig. 5. An example of an e-Government process for obtaining a wheelchair.

To follow the second road to law implementation, a number of key ingredients are necessary. First, the law interpretation document has to be as close as possible to the implementation level or, more realistically, someone has to translate the interpretation law document into some kind of workflow language (cf. Figure 5) using standard ontologies (e.g., Breuker et al. (2003)). Second, the implementing body must have a Service-oriented organization and ICT architecture (cf. the discussion about the Italian case in Mecella and Pernici (2001); Baldoni et al. (2008)). In fact, once the process implementing the law is in place, it will have to invoke services available in the municipality to complete its execution. These may be both performed by a software element or a human being. Third, one needs to have a way to express the generic process describing the law and a framework for the adaptation of this to the implementing parties. The process must be unambiguous and as general as possible, while the adaptation must be as easy and automatizable as possible.

This example is the object of a separate study in the context of the Dutch project Software As Service for the varying needs of Local eGovernments (Aiello et al., 2008).

2.3 Domotics

Domotics is the field where housing (domus) meets technology in its various forms (informatics, but also robotics, mechanics, ergonomics, and communication) to provide better homes from the point of view of safety and comfort. The typical situation of any home is that many heterogeneous devices populate it (Aiello and Dustdar, 2008). Nevertheless, people run similar processes, just using different tools. For instance, one may use a microwave to warm water for a tea while somebody else might use a gas stove. As home appliances are becoming ready for internetworking and interoperation, home human-driven processes can be (semi-)automatically managed and new home software products can emerge.

Since homes are different in the devices that populate them, there is a definite need to model variation when designing home products. Variation points are

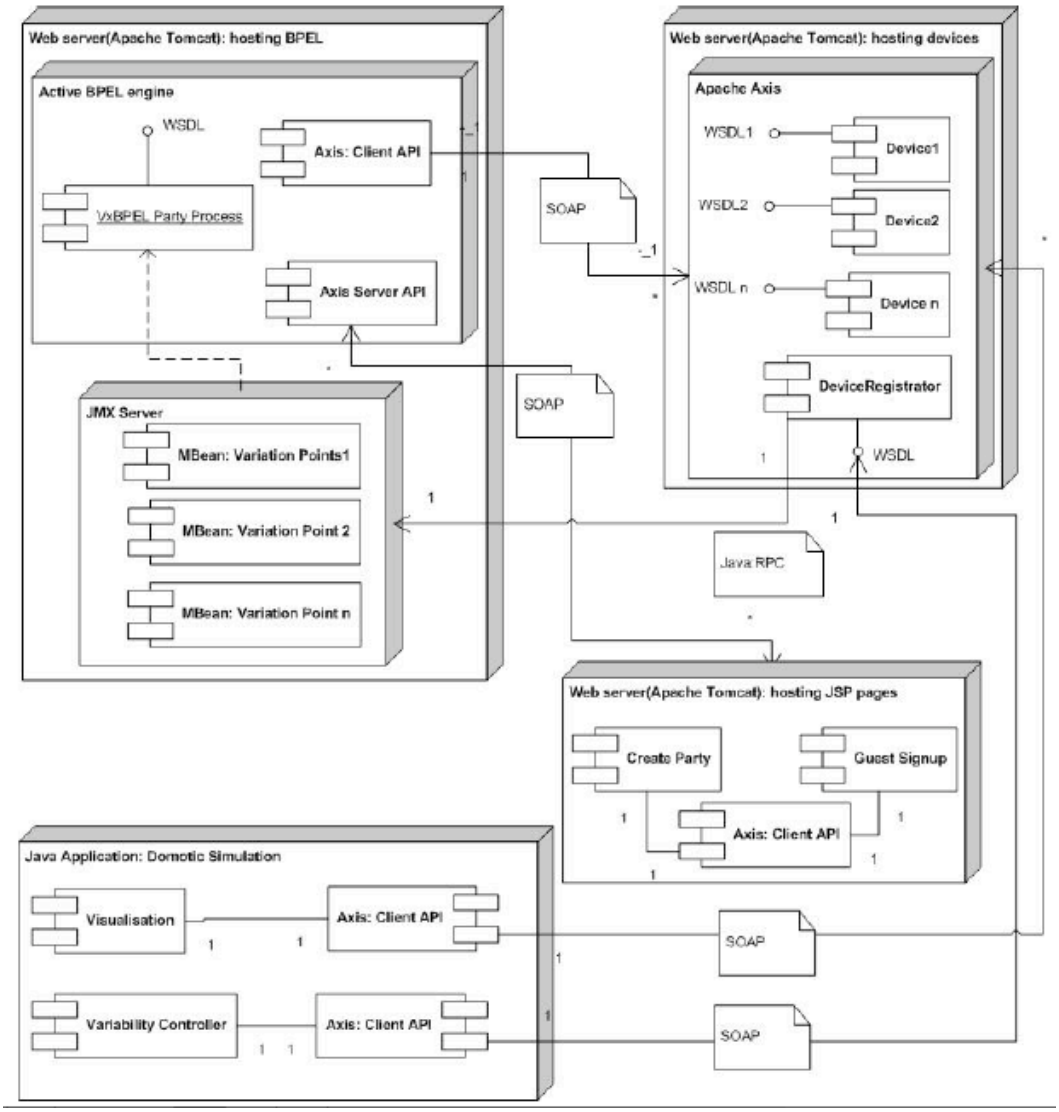

Fig. 6. The architecture of a generic Web service-based domotic middleware (den Dulk, 2009).

then necessarily instantiated at run-time in a specific home. In (den Dulk, 2009), we model the process of organizing a house party. The modeling takes into account various different homes and the possibility that of having different devices or that the same type of device has varying capabilities. Despite these differences, the same process runs in the homes. This is possible when the home devices are available as Web services. den Dulk (2009) proposes a Web service-based architecture based on VxBPEL, illustrated in Figure 6. In the architecture, we remark the component controlling the variability in the BPEL engine (top left), the devices exposed as WSDL interfaces (top-right), and the variability controller at the home level currently based on a visualization and simulation (ViSi) environment (bottom-left). The ViSi tool is described in Lazovik et al. (2009).

3 Background

The COVAMOF framework, VxBPEL, and the UML extension mechanisms represent the concepts and techniques used by our architectural variability modeling approach.

3.1 The COVAMOF framework

In (Sinnema et al., 2004), we have shown why existing variability modeling approaches are inadequate to handle the variability issues relevant for industrial purposes. In response, the COVAMOF framework was proposed to assist developers in the modeling and managing of variability in software product families (Sinnema et al., 2004, 2006a,b; Deelstra et al., 2005).

The COVAMOF framework offers modeling facilities to model variation points and dependencies uniformly over multiple layers of abstraction, i.e., the feature layer, architectural layer, and implementation layer. Dependencies are system properties whose value is influenced by the selection of variants at variation points. Variation points and dependencies are modeled as first-class citizens, which means they are explicit entities in the model, and can be used without restriction.

Part of the COVAMOF framework is the COVAMOF-VS tool suite, which is an add-in for Microsoft Visual Studio .NET. The tool suite can be used to create variability models of a software product family, and these models can then be used for the derivation of individual products.

COVAMOF enables providing different views on the variability within a product family. At the moment, it supports two views: the *variation point view* and the *dependency view*. This separation of views is possible thanks to variation points and dependencies being treated as first-class citizens.

The variation point view shows which choices are available at the different layers of abstraction. It also shows how these choices realize each other across layers. This view contains the following entities: variation point, variant, realization, and dependency. These entities are described in more detail below. Using the variation point view, an engineer can configure individual products.

The dependency view shows how the dependencies interact with each other, and it shows how to deal with these interactions. It contains the following entities: dependencies and dependency interactions, which are explicitly part of the variability model.

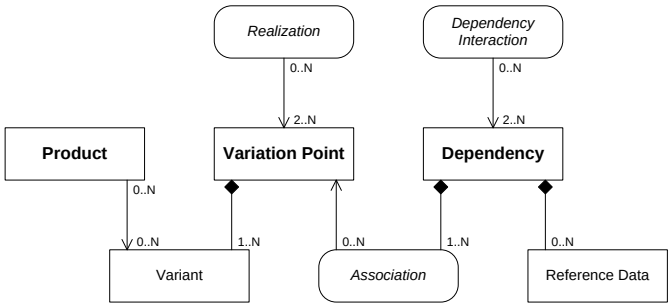

Fig. 7. The COVAMOF meta model.

In order to provide the different views, the tool suite maintains an integrated variability model. Variability information is extracted from the files in the active Solution in Visual Studio, which contains the artifacts of software product family. All COVAMOF models conform to the COVAMOF meta model, which is presented in Figure 7.

The different variability concepts within the COVAMOF meta model are the following ones.

- *Variation point and variant.* Variation points represent a location at which a choice is provided. A variation point has a number of properties, such as variation type, abstraction layer, binding time, and rationale. Variants represent the options available at a variation point. Variants have an effectuating actions property, which specifies which effectuating actions should be executed when the variant is selected.
- *Realization.* Variation points can exist at different layers of abstraction. Realization relations specify rules that determine which variants at lower layers of abstraction should be selected, in order to realize the choice at variation points in higher layers.
- *Dependency.* A dependency represents a system property and specifies how the binding of variation points influences the value of that property, i.e., how the selection of certain variants influences the value of that property. Dependencies can have many variation points from different layers of abstraction associated with it and bridge multiple artifacts.
- *Association.* For each variation point associated to a dependency, an association entity is part of the dependency. Associations refer to variation points that affect the value of the system property. Each association defines the relation with one variation point.
- *Reference data.* Besides associations, dependencies also contain so-called reference data elements. These entities contain information on the value of the system property acquired through testing. They consist of a set of variation point bindings, and the corresponding value of the system property.

3.2 VxBPEL

VxBPEL (Koning et al., 2009) is an extension to the process description and definition language BPEL that allows for run-time variability and variability management in Web service-based systems. It contains additional XML elements that store the relevant variability information in a BPEL file. Variability information is defined inline in the process definition. This means adding the variability information as extension elements inside the process definition itself, using a different namespace. This is in fact the recommended way to extend an XML format like BPEL. The elements added to BPEL allow for capturing the four different types of variability listed in (Topaloglu and Capilla, 2004). Variation points can be added to the BPEL code, to support service replacement, different service parameters, and changing the system composition (Sun and Aiello, 2008).

To test whether VxBPEL works, i.e., if it can make processes variable, the ActiveBPEL engine (ActiveBPEL Web site, 2007) was used. ActiveBPEL is a tool that can read BPEL files and run the described processes. Modifications to the tool were necessary to make it parse VxBPEL files, and to handle the additional elements.

For managing the variability of the system externally at run-time JMX was used. JMX (Java Management eXtensions) (Java Management Extensions Web site, 2007) is a tool that explicitly exposes the functionality of objects, in order to monitor and manage them. It was shown that using the modified ActiveBPEL engine in combination with JMX makes it possible to run a VxBPEL process and manage it at run-time. The COVAMOF framework was used to provide an overview of the variability within the Web service-based system.

3.3 The UML extension mechanisms

The Unified Modeling Language is defined within a four layer meta-modeling architecture. The top level is the meta-meta model layer, which defines a language to construct the meta model layer. The meta model layer defines how the UML models, i.e., the model layer, are constructed. Below the model layer, there exists the user objects layer, which is used to construct specific instances of a given model (Medvidovic et al., 2002; Sun, 2002; Sun et al., 2003).

The architecture of a Web service-based system is described using the model layer, with the meta model layer defining how the models should be specified. So, our extensions to UML, which in the UML specification is called a *profile*,

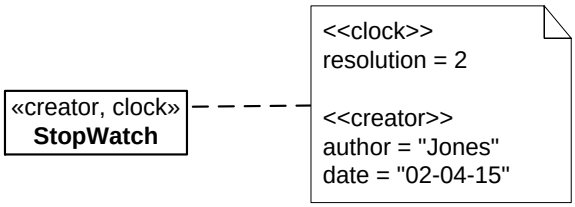

Fig. 8. Stereotypes and tagged values.

are defined by extending the meta model layer.

In UML, there are a number of language extension mechanisms to customize and extend the semantics of model elements, i.e., *constraints*, *tagged values*, *stereotypes*, and *profiles*.

- *Constraints* place added semantic restrictions on model elements. They are denoted as `constraint description`. The constraint description can be in any format, whether it be predicate calculus or natural language.
- *Tagged values* are used to extend modeling elements with extra information. A tagged value is a pair consisting of a name (the tag) and a value, denoted as `tag=value`. Model elements can have an unlimited number of tagged values. The value part of a tagged value can have a special interpretation, such as string, number, or Boolean value.
- *Stereotypes* allow groups of constraints and tagged values to be given descriptive names, and applied to model elements. In this way, a new restricted form of a meta class can be created, which can be used to construct models. A stereotype is denoted by its name between $\ll$ and $\gg$. Any model element, such as a class or a relationship, can have a stereotype attached to it. An example of the use of stereotypes and tagged values is presented in Figure 8.
- *Profiles* are predefined sets of stereotypes, tagged values, and constraints to support modeling in specific domains.

4 A COVAMOF compatible UML profile for modeling architectural variability of Web service-based systems

To model variability in the architecture of Web service-based systems we first define a profile for the Unified Modeling Language (UML). This profile allows us to model COVAMOF variability concepts in individual UML diagrams, and to make variation points span over multiple UML diagrams.

We use UML, because it is very widely used for modeling software architectures, has a straightforward graphical notation, and provides good extension mechanisms. And, by making our profile compatible with the COVAMOF framework, we make sure we can leverage COVAMOF' full potential for vari-

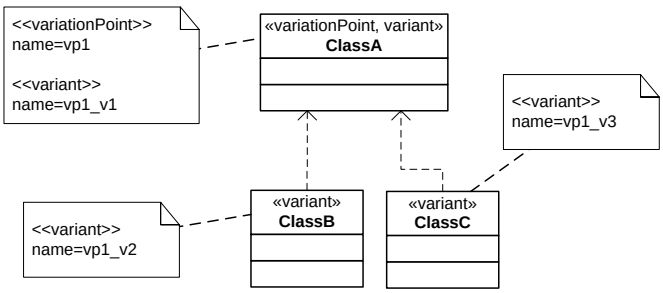

Fig. 9. Variation point of type *C1*.

ability management.

To be compatible with the COVAMOF framework, the following variability concepts are supported: variation point, variant, realization, dependency, association, and reference data. Of these concepts, variation point and variant are modeled directly in the individual diagrams, while the other concepts are modeled by a separate UML diagram, the *Variation point Interaction Diagram (VID)*.

First, in Section 4.1 we define the extensions needed for variation points and variants in class, activity, sequence, and deployment diagrams. Then, in Section 4.2, we describe the Variation point Interaction Diagram.

4.1 Variation point and variant

4.1.1 Class diagrams

Class diagrams show the object-oriented relationships among classes. In class diagrams, the following types of variability are possible:

C1 Selecting a class at a specific position in the diagram.
C2 Selecting an association at a specific position in the diagram.

A variation point with a choice between multiple classes at a specific position in the diagram (*C1*) can be modeled as shown in Figure 9. The location of the variation point is marked with the stereotype ≪*variationPoint*≫. The class with the ≪*variationPoint*≫ stereotype is always one of the variants, which are marked with the ≪*variant*≫ stereotype. In this case, there is a choice between three classes: `ClassA`, `ClassB`, or `ClassC`.

Variation points and variants have a number of attributes, but they are not stored in this diagram. Variation points can span over multiple diagrams. Therefore, it is desirable to place the attribute values in a central place, to avoid inconsistencies. Variation points and variants in this diagram only

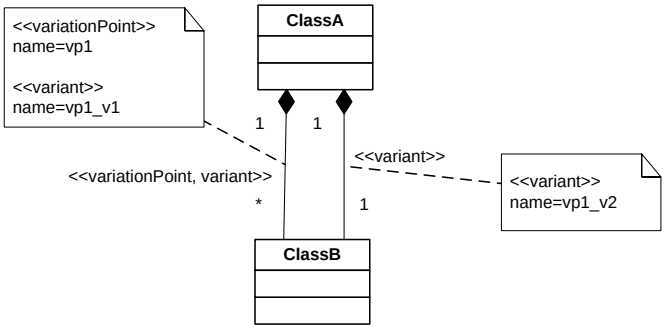

Fig. 10. Variation point of type *C2*.

have a name attribute, through which they are referenced. Classes that are ≪*variant*≫ but not selected for a variation point, are semantically not present in the diagram.

Selecting an association at a specific position in the diagram (*C2*) is modeled as in Figure 10. In this case, the choice is between two composition associations. The associations are both present in the diagram, but marked with the ≪*variant*≫ stereotype. The first variant also holds the ≪*variationPoint*≫ stereotype. The attributes are defined similarly to Figure 9.

Making a class or association optional can be done by creating an optional variation point with just one variant. So, for this type of variability no extra semantics is needed.

4.1.2 Activity diagrams

Activity diagrams focus on the flow of activities involved in a single process. Two types of variability can be found in an activity diagram:

A1 Selecting a particular path within the diagram at a specific position. This is different from simply using a branch element: only the paths of selected variants are semantically present in the diagram.
A2 Selecting an entire swimlane (partition) in the diagram, but keeping the elements within the swimlane unaltered.

A variation point for selecting a path in an activity diagram (*A1*) can be modeled using the fork and join model elements. An example of a variation point with a selection between two paths is shown in Figure 11. A fork element with the ≪*variationPoint*≫ stereotype denotes the beginning of the variation point. A join element with the ≪*variationPointEnd*≫ stereotype denotes the end of the variation point. The variants are paths from the fork to the join. Variants are marked with the ≪*variant*≫ stereotype on the first transition of the path. If a variant is not selected, the entire path is semantically not present in the diagram. All paths of a variation point have to join at the

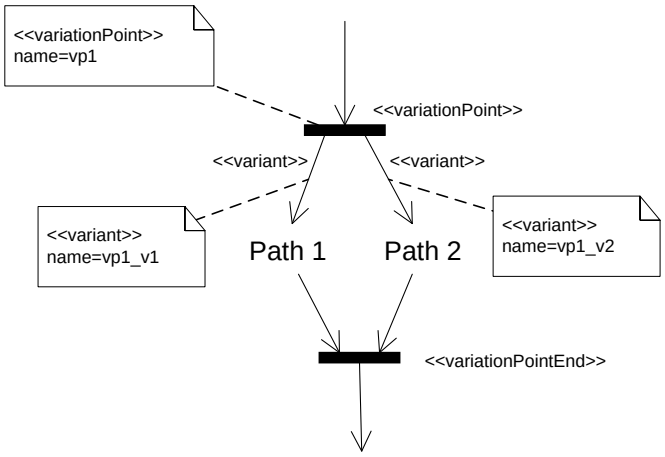

Fig. 11. Variation point of type *A1*.

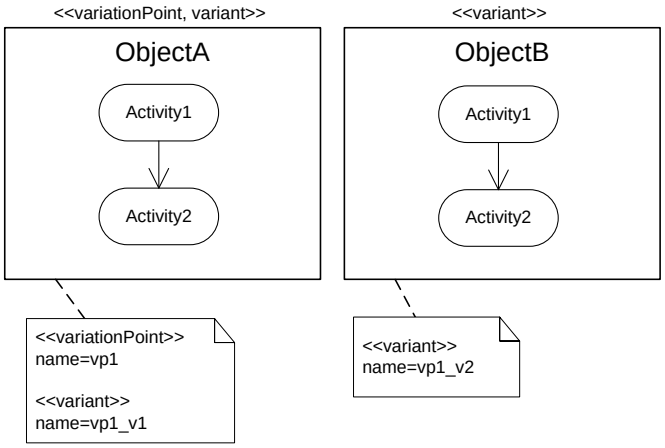

Fig. 12. Variation point of type *A2*.

≪*variationPointEnd*≫. There can be no disjoint paths. However, it is possible to define a new variation point within a variant. All paths of this variation point must end before the enclosing variation point ends.

Selecting an entire swimlane in the diagram (*A2*) can be modeled as shown in Figure 12. The swimlane under selection has the ≪*variationPoint*≫ as well as the ≪*variant*≫ stereotype. The alternative swimlanes are marked with ≪*variant*≫ stereotypes. If a variant partition is not selected, it is semantically not present in the diagram.

4.1.3 Sequence diagrams

A sequence diagram describes interactions between objects, by detailing what messages are sent and when. Sequence diagrams are organized according to time. In a sequence diagram, the following types of variability are possible:

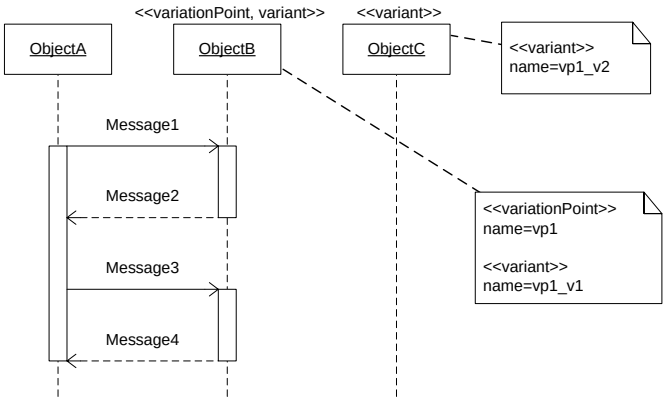

Fig. 13. Variation point of type *S1*.

S1 Selecting an object at a specific position in the diagram.
S2 Selecting a message at a specific position in the diagram.

A selection between different objects (*S1*) can be modeled as shown in Figure 13. The object elements are marked with the ≪*variant*≫ stereotype. The object in the position of the variation point is also marked with the ≪*variationPoint*≫ stereotype. Attributes of the variation point and variants are modeled in the usual manner. In this example, the variation point decides which object should be in the position of `ObjectB`; the selection is between `ObjectB` and `ObjectC`. Only selected lifelines are semantically present in the diagram.

A selection between different messages (*S2*) can be modeled in a similar fashion. Figure 14 shows an example of this. In this case, the selection is between `Message1a` and `Message1b`. The message arrows are marked with the ≪*variant*≫ stereotype, and one of them also with the ≪*variationPoint*≫ stereotype. Of the ≪*variant*≫ messages, only the selected messages are semantically present in the diagram.

4.1.4 Deployment diagrams

Deployment diagrams show how artifacts are distributed over different locations in a network. In a deployment diagram the following types of variability can be identified:

D1 Selecting an artifact at a specific position.
D2 Selecting a communication path between artifacts.
D3 Selecting a node at a specific position.

Selecting an artifact at a specific position in the diagram (*D1*) is modeled as shown in Figure 15. The modeling of a variation point in this diagram is similar to modeling a variation point in a class diagram.

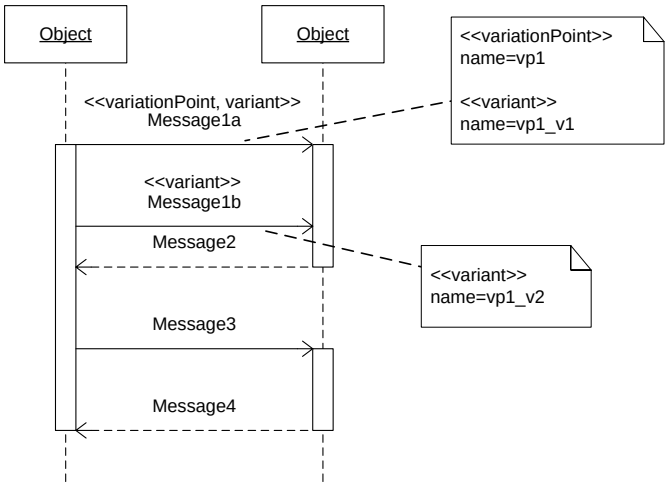

Fig. 14. Variation point of type *S2*.

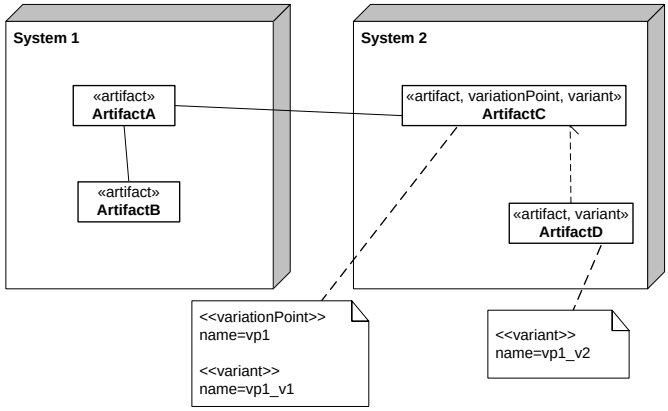

Fig. 15. Variation point of type *D1*.

Selecting a communication path at a specific position in the diagram (*D2*) is modeled analogous to associations in class diagrams. Figure 16 shows how to model this type of variability in a deployment diagram.

Finally, selecting a node at a specific position in the diagram (*D3*) is modeled as displayed in Figure 17. This modeling is similar to modeling partitions in an activity diagram.

4.2 Variation point Interaction Diagram

Modeling variation points and variants in individual UML diagrams is not sufficient, as we also want to model the concepts such as realization, dependency, association, and reference data. Dependencies can have many variation points from different layers of abstraction associated with it and bridge multiple artifacts. Therefore, it is not logical to model dependencies directly in specific

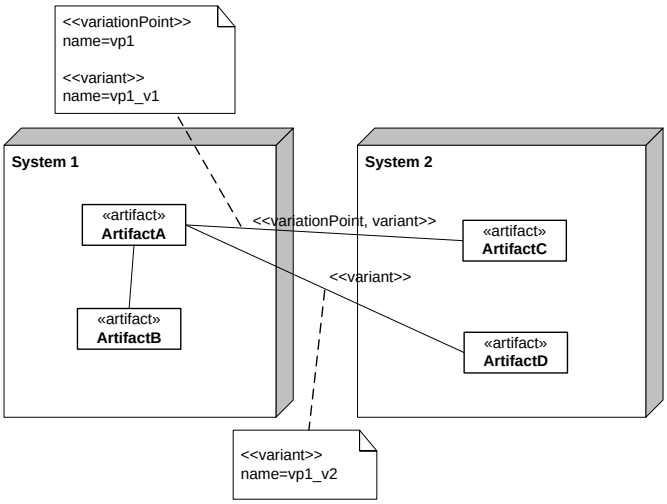

Fig. 16. Variation point of type *D2*.

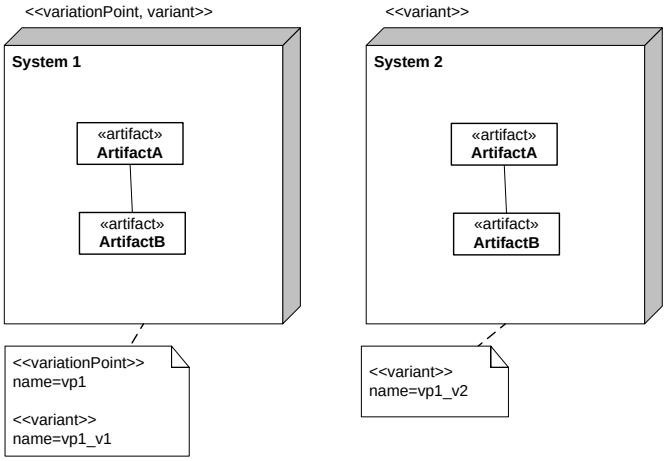

Fig. 17. Variation point of type *D3*.

software artifacts. Besides, it is impossible to add dependencies in all possible UML diagrams, because they are separate entities.

Therefore, we create an additional artifact, the *Variation point Interaction Diagram (VID)*. This is an extended UML class diagram that models the interaction between variation points. In this diagram, all variation points, variants, dependencies, associations, and reference data elements are modeled. Also, all the attributes of these entities are stored. The VID is part of our UML profile for variability; it is a required element of any Web service-based system architecture that supports variability.

This approach has the following advantages. A clear visual overview of the variability is provided in the software artifacts, making adding and editing dependencies easier. Also, in this way the attributes of variation points and variants are stored in a central location, which allows variation points to span

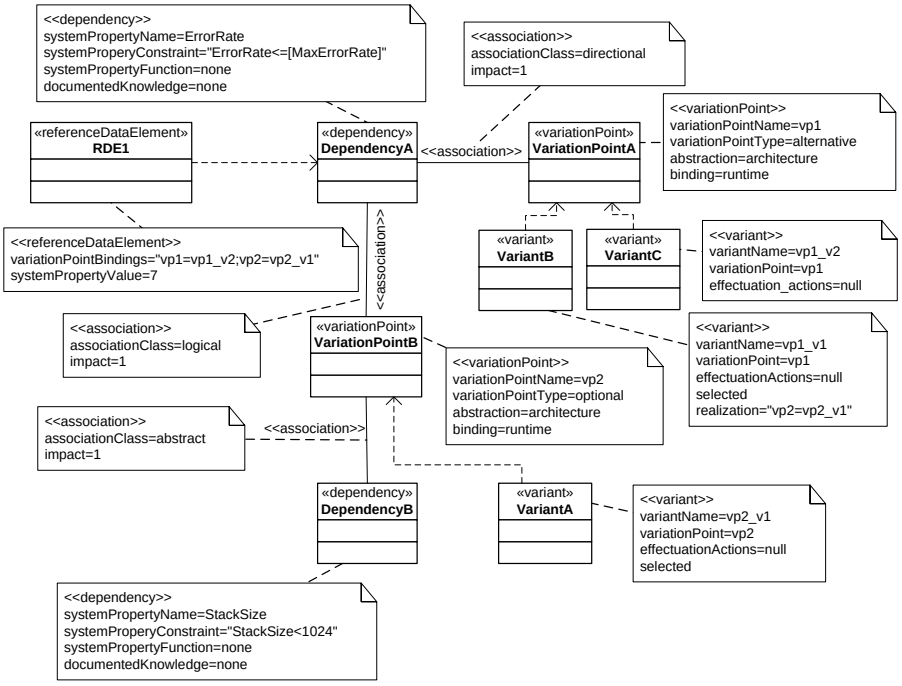

Fig. 18. A Variation point Interaction Diagram.

over multiple diagrams.

An example of the modeling of the Variation point Interaction Diagram is shown in Figure 18. This example shows all the elements a VID can contain and how they are modeled. Realization relations can be modeled in the VID as an attribute of variants. This realization attribute has as its value a list that defines the required bindings of variation points in lower layers of abstraction in order to realize *this* variant.

A dependency entity can be associated to a variation point entity through an $\ll association \gg$ relation. A reference data element entity is linked to a dependency entity by a UML dependency relation. A variant entity is linked to its variation point entity also by a UML dependency relation. All stereotypes have the attributes that are defined by the COVAMOF framework.

5 Modeling variability of Web service-based systems using the UML profile

There are a number of different types of variability possible in Web service-based systems that we need to support. In (Koning et al., 2009) we already listed four of these types. Here, we add the ability to create complex dependencies:

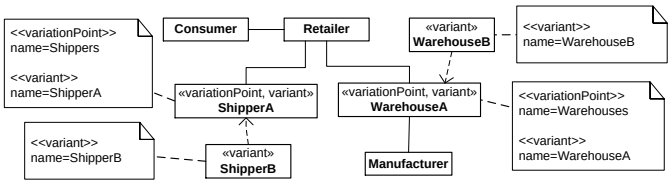

Fig. 19. Composition view with variability.

T1 Replacing a service by a different service with the same interface.
T2 Replacing a service by a different service with a different interface.
T3 Changing the parameters with which a service is invoked.
T4 Changing the composition of the service-based system.
T5 Creating complex dependencies.

In the following, we describe for each type of variability how it can be modeled over the different architectural views of the example system we described in Section 2 and how these views are interconnected.

Suppose there are two shippers in the SCMS, `ShipperA` and `ShipperB`, which both are represented by a Web services with exactly the same interface (*T1*). There is variability in the SCMS architecture in the sense that there is a selection between these two shippers. For this variability, a variation point `Shippers` is added, which appears also in the different views of the architecture. The variation point has two variants `ShipperA` and `ShipperB`, which represent the choice between `ShipperA` and `ShipperB`.

In the composition view of the SCMS architecture, the variation point `Shippers` is modeled as shown in Figure 19. The shipper Web services are also involved in the business process view, so variation point `Shippers` is also present in that view; it can be modeled as shown in Figure 20. `ShipperB` implements the same interface as `ShipperA`, so the same activity can be performed by both Web services without difficulty. In the use case scenario view, variation point `Shippers` is modeled as shown in Figure 21. `ShipperA` can be replaced without difficulty by `ShipperB`, because they have the same interface. In the deployment view, i.e., in the deployment diagram, variation point `Shippers` is also present. It is modeled, as displayed in Figure 22. Variation point `Shippers` is now present in all the necessary views. What remains, is modeling the Variation point Interaction Diagram, which stores the attributes of the variation points and variants. This diagram, which is mandatory, is modeled in Figure 23.

Now suppose a retailer in the SCMS has access to two warehouses, `WarehouseA` and `WarehouseB`, but the warehouses are represented by Web services with different interfaces (*T2*). We add a variation point `Warehouses` to the SCMS architecture for selecting one of the two warehouses. This variation point has two variants, `WarehouseA` and `WarehouseB`.

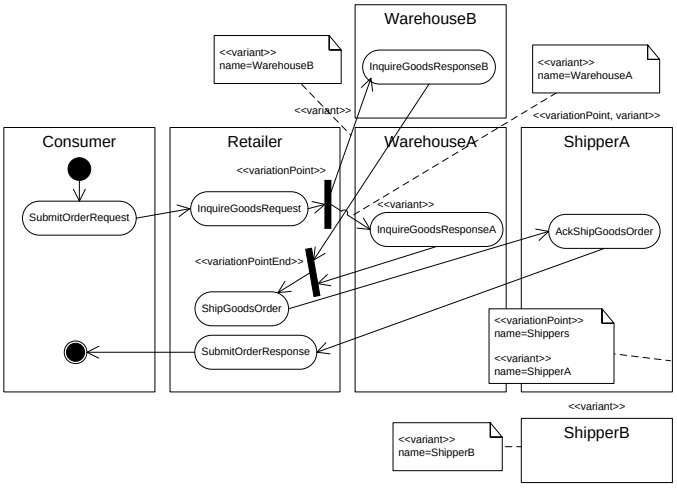

Fig. 20. Business process view with variability.

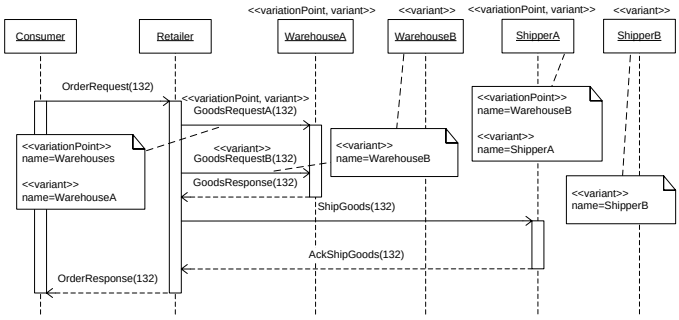

Fig. 21. Use case scenario view with variability.

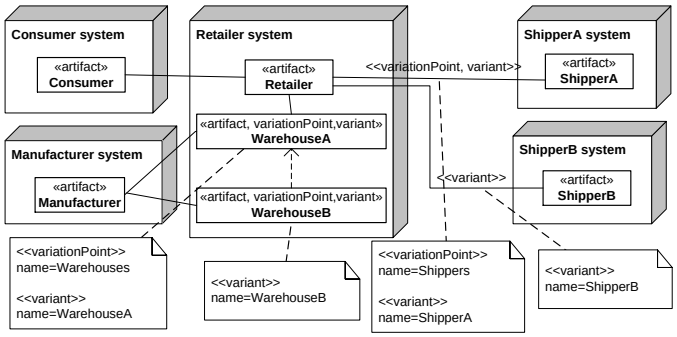

Fig. 22. Deployment view with variability.

In the composition and deployment views (Figures 19 and 22) variation point `Warehouses` can be modeled in the same way as `Shippers`, because the interface plays no part in those views. However, in the business process and use case scenario views (Figures 20 and 21) we need to model variation point `Warehouses` differently, because a different exchange of messages is necessary. Variation point `Warehouses` is also present in the VID (Figure 23).

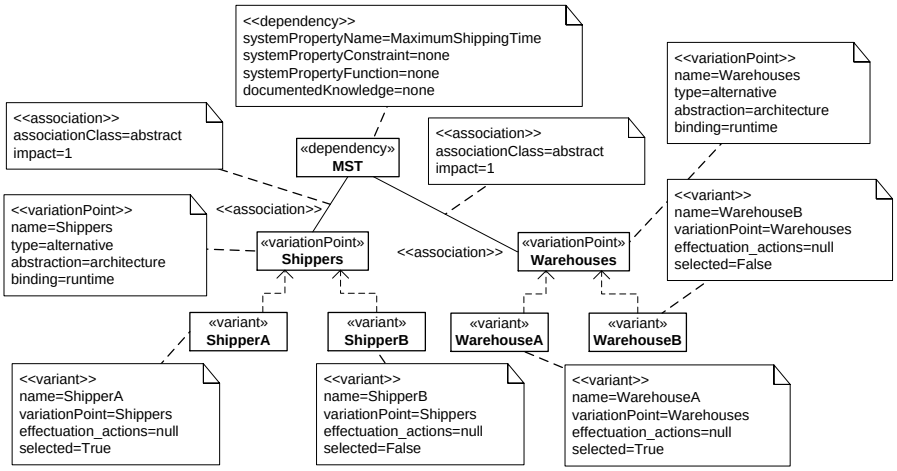

Fig. 23. The VID for the SCMS architecture.

Invoking a Web service with different parameters (*T3*) can be seen as sending a different message to it. How to model a variation point for selecting a message is already illustrated by variation point `Warehouses`. The only difference is that there is no selection between services. This type of variability only affects the business process and use case scenario views (Figures 20 and 21).

Changing the composition of a Web service-based system (*T4*) means replacing a set of interconnected Web services by a different set of interconnected Web services. Variation points `Shippers` and `Warehouses` already show how it is done for one Web service. Replacing a larger part of the system in the composition view can be done by following some rules. Any class that is part of the variant is marked with the ≪*variant*≫ stereotype. When a variant is replaced, all classes of the variant and any interconnecting associations are replaced by the other variant. The head of the variant is defined by the position of the variation point. The names of the classes in the variants are used to connect outward associations, if there are any.

The last type of variability we support is the modeling of complex dependencies (*T5*). A dependency represents a system property and specifies how the binding of variation points influences the value of that property. Complex dependencies are the result of the combination of variants for various variation points. The COVAMOF framework supports the modeling of complex dependencies, and since our UML profile is compatible with COVAMOF, we can model complex dependencies in UML. The VID supports all the concepts needed to do this.

Suppose there is a dependency `MST` that represents the system property *maximum shipping time*, which is the maximum time it takes to deliver an ordered good to the consumer. The value of this dependency depends on the selection of variants at variation points `Shippers` and `Warehouses`. We model this

dependency as shown in the VID in Figure 23.

6 Run-time variability management

The behavior of a Web service-based system is completely defined by the software artifacts of the implementation layer. The variability in these software artifacts is viewed using the COVAMOF-VS tool suite. The tool suite is also used to alter the software artifacts through their variation points. Owing to our UML profile for architectural variability, we can model variability in all three layers of abstraction. Using the COVAMOF-VS tool suite, a user reconfigures variation points, which can be in any abstraction layer. Then, through realization relations, COVAMOF-VS automatically configures the variation points in lower layers of abstraction. The COVAMOF-VS tool suite uses so-called model providers to keep the software artifacts consistent with the new binding. After the variation points in the implementation layer have their new binding, the change can also be effectuated to the actual system at run-time. For this, we use the strategy developed in (Koning et al., 2009), i.e., reconfiguring the system by using VxBPEL and Java Management eXtensions (JMX).

The process of managing variability in Web service-based systems at run-time using COVAMOF and UML consists of the following steps.

(1) Create the feature layer of the system using XVL files. XVL is an XML-based language for modeling variability concepts, developed as part of the COVAMOF framework. The feature layer contains variation points that describe the general settings of the system.
(2) Architecture:
    (a) Create the architectural layer of the system by creating UML diagrams that describe its composition, business process, use case scenario, and deployment views. Store these diagrams in the XML Metadata Interchange (XMI) format.
    (b) Create variation points, dependencies, and other entities in the architectural diagrams, to model the required variability.
    (c) Create the Variation point Interaction Diagram (VID) containing all the added entities. Add the required realization relations between the variation points in the feature layer and the architectural layer.
(3) Implementation:
    (a) Create the implementation layer of the system by implementing the individual Web services using any technology, and the BPEL processes that orchestrate the interaction between them.
    (b) Create variation points in the VxBPEL code of the system, to allow for the required variability at the implementation layer. Add the required realization relations between the variation points in the ar-

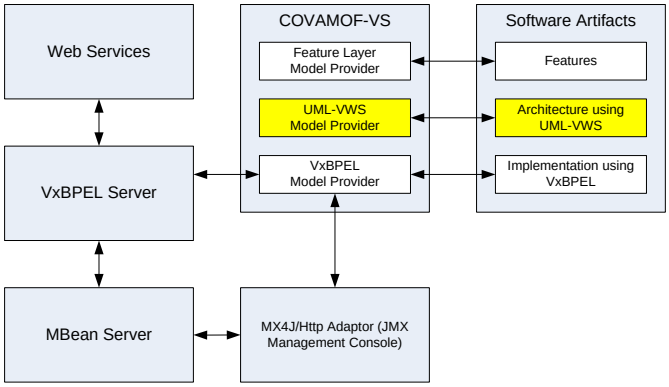

Fig. 24. Participants in the process.

chitectural layer and the implementation layer.

(4) Deploy and run the system, i.e., deploy the individual Web services, and deploy the VxBPEL server using the VxBPEL files.
(5) Deploy an MBean server, which is needed for step 8. Let the variation points in the VxBPEL server register their JMX interface at the MBean server.
(6) Use the COVAMOF-VS tool suite to view the variability in the system, and to reconfigure the variation points and dependencies to the user's wishes.
(7) Effectuate the new configuration using COVAMOF-VS. The variation points in some layer of abstraction are configured, which leads automatically, through the realization relations, to a configuration of the variation points in lower layers of abstraction. The model providers effectuate the configuration of all the variation points back to the software artifacts.
(8) Now that the VxBPEL files are changed, follow the steps described in (Koning et al., 2009) to reconfigure the Web service-based system at run-time. These are:
    (a) Effectuate the new configuration of the variation points in the implementation layer, i.e., in the VxBPEL process, by invoking the MX4J/Http Adaptor (JMX Management Console) (MX4J Web site, 2007).
    (b) The MX4J/Http Adaptor reconfigures the VxBPEL process in the VxBPEL server through the MBean server.
(9) Repeat steps 6 – 8 whenever the system needs to be reconfigured.

Figure 24 shows a diagram of the participants in the variability management process. The arrows indicate interaction between the participants. The participants in the variability management process are:

- *COVAMOF-VS* is the Visual Studio add-in that is used to view and configure the variability in the Web service-based system. No changes to COVAMOF-VS itself are required; only its model providers require changes.

- The *Feature Layer Model Provider* translates between the XVL files of the feature layer and the COVAMOF model. This model provider already exists and does not need to be changed for this process.
- The *UML-VWS Model Provider* is a model provider we developed. It parses the variability information in the architectural diagrams that make use of our UML profile for variability. It reads variation points, variants, dependencies, associations, and realization relations, and uses these entities to create a COVAMOF model. It also effectuates changes back to the XMI file. This model provider is used to manage the variability in the architectural layer.
- The *VxBPEL Model Provider* is the model provider developed in (Koning et al., 2009). At present, it can only extract variability information from a VxBPEL process to create a COVAMOF model. For our management process, this model provider should be extended to allow it to:
  · Effectuate changes in the COVAMOF model back the VxBPEL file.
  · Automatically configure a VxBPEL server. The server should initially run the VxBPEL process that is read by the model provider.
  · Automatically invoke the MX4J/Http Adaptor in order to effectuate changes in the variability to the VxBPEL server, i.e., to reconfigure the system at run-time.
- *MX4J/Http Adaptor* is a tool that communicates with the MBean Server to configure variation points through their JMX interface. The VxBPEL model provider should use this tool (without user intervention) to reconfigure the Web service-based system. So, the variability in the implementation layer is viewed in COVAMOF-VS by parsing the VxBPEL files, but the effectuation of a new configuration is performed by invoking the MX4J/Http Adaptor.
- The *MBean Server* is used to reconfigure the variation points in the VxBPEL Server through their JMX interfaces, which they will register at this server. The MX4J/Http Adaptor is used to control this server.
- In the *VxBPEL Server*, which is a BPEL server adapted for VxBPEL, each variation point in the VxBPEL process registers their JMX interface at the MBean Server. The VxBPEL model provider communicates with this server to set up the initial VxBPEL process. The VxBPEL Server controls the Web service-based system through the VxBPEL process.
- The process that the *Web services* execute is managed by the VxBPEL Server.

7 Implementation

With the process and tools developed by us, one can automate the variability of Web service-based systems. We below show major artifacts of the SCMS using our approach.

The UML diagrams used in the architecture of the Web service-based system

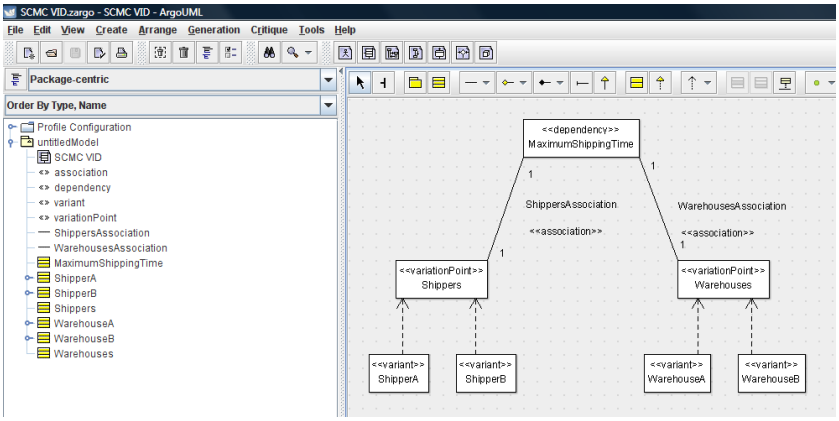

Fig. 25. The VID of the SCMS in ArgoUML.

have to be stored in a machine readable format, in order for a model provider to extract the COVAMOF variability model from it. Therefore, we use the XML Metadata Interchange (XMI) (XML Metadata Interchange (XMI) Web site, 2007) format to store the UML diagrams (Sun et al., 2003). This is an OMG standard for exchanging metadata information via XML. The most common use of XMI is as an interchange format for UML models, and it is supported by many UML tools.

We use ArgoUML (ArgoUML Web site, 2007) for creating the UML diagrams that are part of the software artifacts. ArgoUML is an open source UML tool, developed in Java. This tool provides all needed features, such as stereotypes and tagged values, and allows for diagrams to be exported to the XMI format.

All the variability information COVAMOF-VS needs, is available in the Variation point Interaction Diagram (VID). All variation points, variants, dependencies, and other COVAMOF entities, and their attributes are stored in this diagram. Therefore, only this diagram needs to be parsed.

To illustrate ArgoUML and XMI, the VID of the SCMS presented in Figure 23 is modeled using ArgoUML and stored in the XMI format. Figure 25 shows a screenshot of the diagram in ArgoUML.

Model providers in COVAMOF-VS are used to extract the variability information from the software artifacts. For artifacts in the feature layer (XVL files), and the implementation layer (VxBPEL files), we already have model providers. For the architectural layer, i.e., the XMI file describing the VID, a new model provider is required.

We developed this model provider, called the UML-VWS Model Provider, which is a DLL file used by COVAMOF-VS. The model provider parses the variability information from an XMI file of a VID. It reads variation points, variants, dependencies, associations, and realization relations, and uses these

entities to create a COVAMOF model.

To effectuate the configuration of variation points in the architecture of a Web service-based system, our model provider alters the XMI file, selecting the correct variants for the variation points.

One can bind a variation point in the feature layer using COVAMOF-VS, which leads automatically to bindings of other variation points in lower layers of abstraction. The UML-VWS Model Provider takes care of the effectuation to the architectural artifacts. By adapting the XMI file that represents the VID, the software artifacts are kept consistent with the COVAMOF model.

8 Related work

Variability Management is an important reuse issue in product families (Linden, 2002). Many variability modeling approaches have been reported (Sinnema and Deelstra, 2007). However, they are not adequate to handle variability issues relavant for industrial purposes (Sinnema et al., 2004). This observation resulted in the creation of the COVAMOF framework, a variability management tool which has been tested and evaluated positively in industrial settings (Sinnema and Deelstra, 2008). Next, we overview and discuss related work on modeling variability of Web service-based systems.

The UML profile described in this paper builds on solid related work. Namely, the way variation points are modeled resembles how Clauß models them (Clauß, 2001b,a). A similar use of stereotypes is also described by Oliveira Junior et al. (2005) . However, there are important differences. The existing methods only model variation points and variants, but not realizations, dependencies, associations, and reference data. We do model all these concepts, in order to be compatible with the COVAMOF framework. The existing methods only model variability in UML class and use case diagrams, while our approach also models variability in activity, sequence and deployment diagrams. In fact, our modeling constructs can be applied to any type of UML diagram. The existing methods define variability for class elements, but not for association elements, while our approach also allows for variability in associations between classes.

In defining variation points and variants in class diagrams there are also some differences with existing work. Clauß (2001 b,a) defines variation points in UML class diagrams, but there are two differences with our approach. In his approach, multiple variation points can be associated with a single class, while in our approach we always replace the entire class to make a change. His approach has the advantage that it can reduce the number of variation points needed. The advantage of our approach is that it is more straightforward. He

uses the ≪*optional*≫ stereotype to define optional classes. In our approach an optional element can be created by defining a variation point with its type attribute set to optional.

Oliveira Junior et al. (2005) also define variation points and variants. In their approach more stereotypes are used, such as ≪*optional*≫, ≪*alternative_OR*≫, ≪*mandatory*≫, ≪*mutex*≫, and ≪*requires*≫. This makes it easier to model some concepts, such as mutual exclusion, but it also requires more extensions to UML. They define variability in UML use case diagrams; we see use case diagrams not as part of the views that are most important for Web service-based systems.

K. Mohan and B. Ramesh (2003) present an approach that makes use of an ontology for variability management in product and service families. An ontology developed to catalogue the different concepts of variability, such as variation points and variants. Interviews with domain experts are used to identify the initial concepts, their properties, and the relationships with other concepts. The ontology contains domain specific concepts as well as more general variability concepts. Such ontology for variability is represented electronically using ontology tools. The ontology is integrated by a Knowledge Management System (KMS) to assist designers of a system in implementing variability. Via the KMS, the ontology can be queried for mechanisms used in past projects or other members of the product that implement a specific type of variability. The advantage of this approach is that it offers flexibility in the use of different mechanisms for implementing variability. Another advantage is that it is domain independent, i.e., solutions from other domains can be used in the current project. However, a drawback is that it requires major involvement from the user. Which in turn it means that unfortunately the approach can not be used for automatic reconfiguration of a system.

S. Robak and B. Franczyk (2003) introduce the concept of modeling the variability of Web services using feature diagrams. Feature diagrams allow for the presentation of the commonalities and variabilities of the concept they describe. A feature is defined as a visible characteristic of a concept, which is used to describe the concept and to distinguish different instances of the concept. A concept can be anything, such as a system or a component. The feature model indicates the intention of the described concept. The set of instances described by the feature model is called the extension of the concept. The feature model can be used to make a generic description for a range of systems. For a Web service-based system a feature diagram can be created describing the commonalities and differences within the range of possible systems. This base can then be used to specify specific systems to meet certain needs. The advantage of this methodology is that it supports automated configuration of a system. Another advantage is that it provides a clear overview of the variability and commonalities within a system. However, describing variability only in this

manner means that realization relations and dependencies are not modeled.

K. Mantell (2005) describes a UML profile to model business processes, and shows how a UML model of a business process can be mapped directly to BPEL code. The profile allows developers to use normal UML skills and tools to developWeb service processes in BPEL4WS. By describing a BPEL process using UML, there is a higher perceived level of abstraction. Using UML to model a BPEL process makes it more comprehensible for humans. However, this approach does not include variability management.

9 Conclusion

We have designed a UML profile for architectural variability modeling in Web service-based systems. This profile is compatible with COVAMOF, because concepts such as variation point, variant, realization relation, dependency, association, and reference data are explicitly modeled in the UML diagrams. In addition, we have described how to use this UML profile to support the five different types of variability in the architecture of a Web service-based system. We did this by describing how the variability is modeled over the different views of the architecture, and how variation points in these views relate to each other through the Variation point Interaction Diagram.

To manage the variability in Web service-based systems at run-time, we have developed a variability management process that requires only minimal involvement from the end-user. This management process is driven from the COVAMOF-VS tool suite extended by us, and makes use of our architectural variability modeling approach.

Through this work, we have brought COVAMOF's enormous potential for variability management in industry to the fast growing world of Web service-based systems. Through explicit variability management, service providers or service composition designers can gain the advantages such as the Quality of Service support, the availability enhancement, the quality attribute optimization, and the run-time flexibility support. Moreover, with the proposed methodology variability can be modeled also at the architectural level, which enables systematic management of the variability.

COVAMOF and VxBPEL are first steps in providing tools that enhance reusability. Two roads appear particularly interesting for future investigation. First, there is the issue of managing the evolution of the variability models. What happens if there is a modification to a model? Can these be translated directly into the instantiated products? This is quite common in the case of eGovernment, where updates and extensions to a law can be frequent and

imply modifications to all local instances of systems implementing the law. Second, there is the issue of managing variability in open environments where variation points are managed by independent actors. Then it is important to add semantic descriptions to these. Ideally, one should be able to entirely automate the task of making variability decisions by selecting the best variation based on semantic description of the pre- and post-conditions associated with it. To make such annotations OWL-S (Martin et al., 2004) is a natural candidate,1 having the appropriate expressive power and being nicely integratable into Web service-based architectures.

Acknowledgements

We thank everybody who has contributed to the implementation of the COVAMOF and VxBPEL frameworks. Without them, this research would not have been possible. The work is supported by the European Union Integrated Project SeCSE (Service Centric Software Engineering, `http://secse.eng.it`) IST Contract no. 511680,, NWO Jacquard project Software as a Service for the varying needs of Local eGovernment (**SaS-LeG**, `http://www.sas-leg.net`), contract no. 638.000.000.07N07; and the Science and Technology Foundation of Beijing Jiaotong University (grant no. 2007RC099).

References

ActiveBPEL Web site, 2007. `http://www.activebpel.org/`.

Aiello, M., Avgeriou, P., Lazovik, A., Wortmann, H., 2008. Software as service for the varying needs of local egovernments (sas-leg).
URL `\url{www.sas-leg.net}`

Aiello, M., Dustdar, S., 2008. Are our homes ready for services? a domotic infrastructure based on the web service stack. Pervasive and Mobile Computing 4 (4), 506 – 525.

ArgoUML Web site, 2007. `http://argouml.tigris.org/`.

Bachmann, F., Bass, L. J., 2001. Managing variability in software architectures. In: ACM SIGSOFT Symposium on Software Reusability. pp. 126–132.

Baldoni, R., Fuligni, S., Mecella, M., Tortorelli, F., 2008. The italian e-government enterprise architecture: A comprehensive introduction with focus on the sla issue. In: Nanya, T., Maruyama, F., Pataricza, A., Malek, M. (Eds.), Service Availability, 5th International Service Availability Symposium, ISAS 2008, Tokyo, Japan, May 19-21, 2008, Proceedings. Vol. 5017 of Lecture Notes in Computer Science. Springer, pp. 1–12.

Baresi, L., Heckel, R., Thöne, S., Varró, D., 2003. Modeling and validation

of Service-Oriented Architectures: application vs. style. In: ESEC/FSE-11: Proceedings of the 9th European Software Engineering Conference held jointly with 11th ACM SIGSOFT international symposium on Foundations of Software Engineering. ACM Press, pp. 68–77.

Breuker, J., Valente, A., Winkels, R., 2003. Use and reuse of legal ontologies in knowledge engineering and information management. In: Law and the Semantic Web: Legal Ontologies, Methodologies, Legal Information Retrieval, and Applications. Vol. 3369. pp. 36–64.

Business Process Execution Language (BPEL) Web site, 2007. http://www.oasis-open.org/.

Chapman, M., Goodner, M., Lund, B., McKee, B., Rekasius, R., 2003. Sample application supply chain management architecture (version 1.01). Web Services Interoperability Organization.

Clauß, M., 2001a. Generic Modeling using UML extensions for variability. Proceedings of the OOPSLA Workshop on Domain-specific Visual Languages, 11–18.

Clauß, M., 2001b. Modeling variability with UML. Proceedings of the 3rd International Conference on Generative and Component-Based Software Engineering (GCSE).

Curbera, F., Duftler, M., Khalaf, R., Nagy, W., Mukhi, N., Weerawarana, S., 2002. Unraveling the Web services web: An introduction to SOAP, WSDL, and UDDI. IEEE Internet Computing 6 (2), 86–93.

de Oliveira Junior, E. A., Gimenes, I. M., Huzita, E. H., Maldonado, J. C., Alencar, P., October 2005. Adding variability management to UML-based software product lines. Tech. Rep. CS-2005-33, David R. Cheriton School of Computer Science.

Deelstra, S., Sinnema, M., Bosch, J., 2005. Product derivation in software product families: a case study. Journal of Systems and Software 74 (2), 173–194.

den Dulk, P., 2009. Variability in bpel as an infrastructure for domotics. Master's thesis, University of Groningen.

Java Management Extensions Web site, 2007. http://java.sun.com/products/JavaManagement/.

Koning, M., Sun, C., Sinnema, M., Avgeriou, P., 2009. VxBPEL: Supporting variability for Web services in BPEL. Information and Software Technology 51, 258–269.

Lazovik, E., Dulk, P. v., Groote, M. d., Lazovik, A., Aiello, M., 2009. Services inside the smart home a simulation and visualization tool. submitted, video available at http://www.youtube.com/watch?v=Fy83Apxi8sA.

Linden, F. v. d., 2002. Software product families in Europe: The Esaps & Café Projects. IEEE Software 19 (4), 41–49.

Mantell, K., 2005. From UML to BPEL: Model driven architecture in a Web services world. IBM developerWorks.

Martin, D., Burstein, M., Hobbs, J., Lassila, O., McDermott, D., McIlraith, S., Narayanan, S., Paolucci, M., Parsia, B., Payne, T. R., Sirin, E., Srinivasan,

N., Sycara, K., November 2004. Owl-s: Semantic markup for web services. URL http://eprints.ecs.soton.ac.uk/12687/

Mecella, M., Pernici, B., 2001. Designing wrapper components for e-services in integrating heterogeneous systems. VLDB J. 10 (1), 2–15.

Medvidovic, N., Rosenblum, D. S., Redmiles, D. F., Robbins, J. E., 2002. Modeling software architectures in the Unified Modeling Language. ACM Transactions on Software Engineering and Methodology 11 (1), 2–57.

Microsoft Visual Studio .NET Web site, 2007. http://www.microsoft.com/vstudio.

Mohan, K., Ramesh, B., 2003. Ontology-based support for variability management in product and service families. In: Proceedings of the 36th Annual Hawaii International Conference on System Sciences (HICSS'03) - Track 3. p. 75.1.

MX4J Web site, 2007. http://mx4j.sourceforge.net/.

Peltz, C., January 2003. Web services orchestration, a review of emerging technologies, tools, and standards. Tech. rep., Hewlett-Packard Company.

Robak, S., Franczyk, B., 2003. Modeling Web services variability with feature diagrams. In: Revised Papers from the NODe 2002 Web and Database-Related Workshops on Web, Web-Services, and Database Systems. Springer-Verlag, pp. 120–128.

Sinnema, M., Deelstra, S., 2007. Classifying variability modeling techniques. Information and Software Technology 49, 717–739.

Sinnema, M., Deelstra, S., 2008. Classifying variability modeling techniques. Journal of Systems and Software 81, 584–600.

Sinnema, M., Deelstra, S., Hoekstra, P., 2006a. The COVAMOF derivation process. In: Proceedings of the International Conference on Software Reuse (ICSR). pp. 101–114.

Sinnema, M., Deelstra, S., Nijhuis, J., Bosch, J., 2004. COVAMOF: A framework for modeling variability in software product families. In: Proceedings of the Software Product Line Conference (SPLC). pp. 197–213.

Sinnema, M., Deelstra, S., Nijhuis, J., Bosch, J., 2006b. Modeling dependencies in product families with COVAMOF. In: Proceedings of the 13th Annual IEEE International Symposium and Workshop on Engineering of Computer Based Systems (ECBS'06). pp. 299–307.

Sun, C., 2002. Contributions to software architecture construction and description and reconstruction. Ph.D. thesis, Beijing University of Aeronautics and Astronautics.

Sun, C., Aiello, M., 2008. Towards variable service compositions using vxbpel. In: Proceedings of the International Conference on Software Reuse (ICSR), Lecture Notes in Computer Science, 5030. Springer-Verlag, pp. 257–261.

Sun, C., Cao, J., Jin, M., Liu, C., Lyu, M. R., 2003. Extendable and interchangeable architecture description of distributed systems using uml and xml. In: Proceedings of APPT 03, Lecture Notes in Computer Science, 2834. Springer-Verlag, pp. 536–545.

Topaloglu, Y., Capilla, R., 2004. Modeling the variability of Web services from

a pattern point of view. Lecture Notes in Computer Science 3250, 128–138.

Unified Modeling Language (UML) Web site, 2007. http://www.uml.org.

Web Services Interoperability Organization (WS-I) Web site, 2007. http://www.ws-i.org/.

XML Metadata Interchange (XMI) Web site, 2007. http://www.omg.org/xmi/.