Architecture-Driven Variation Analysis for Designing Cloud Applications

Liang-Jie Zhang [1], Jia Zhang [2]
[1]*IBM T.J. Watson Research Center, USA*
[2]*Department of Computer Science, Northern Illinois University, USA*
[1] *zhanglj@us.ibm.com,*[2] *jiazhang@cs.niu.edu*

Abstract

Service Oriented Architecture (SOA) is one central technical foundation supporting the rapidly emerging Cloud Computing paradigm. To date, however, its application practice is not always successful. One major reason is the lack of a systematic engineering process and tool supported by reusable architectural artifacts. Toward this ultimate goal, this paper proposes a variation-oriented analysis method of performing architectural building blocks (ABB)-based SOA solution design for enabling cloud application design. We present the modeling of solution-level architectural artifacts and their relationships, whose formalization enables event-based variation notification and propagation analysis. We report a prototype tool and describe how we extend the Unified Modeling Language (UML) mechanism to implement the system and enable solution-level variation analysis and enforcement in business cloud as an example.

1. Introduction

Service Oriented Architecture (SOA) is one central technical foundation [1] supporting the rapidly emerging Cloud Computing paradigm, a scalable services delivery and consumption platform in the field of Services Computing [2]. It is a powerful model that allows engineers to dynamically integrate and compose existing service components or services into new cloud services. To date, however, the actual SOA-based application practice is not always a success, because most of SOA practices are conducted at an *ad hoc* manner, mainly based on practitioners' personal experiences. Therefore, our research aims to develop a systematic engineering process and an integrated design tool to facilitate SOA-based variation analysis for cloud application design. Although Cloud Computing comprises various levels of cloud services (infrastructure cloud, software cloud, application cloud, and business cloud), without losing generality, in this research, we focus on SOA at application cloud level. We study how to support and facilitate the design and development of SOA-based applications, that is, SOA solutions.

From industry best practice, layered architectural models have been widely adopted by SOA practitioners to build SOA solutions. For example, IBM has proposed Service-Oriented Solution Stack (S3), a layered enterprise architectural template, as a guidance for IT architects to design the overall architecture of an SOA solution [3]. Nine layers are identified and organized into a two-dimensional model. The horizontal dimension (Operational System layer, Services Component layer, Service layer, Business Process layer, and Service Consumer layer) implements functional requirements; and the vertical dimension (Integration layer, Data Architecture layer, QoS layer, and Governance layer) provides system-support facilities and enablement.

To provide a uniform mechanism for building configurable and reusable SOA solutions on top of S3, we introduced a set of usable Architectural Building Blocks (ABBs) as the fundamental units of an SOA solution [2, 4-6]. An ABB is a component that encapsulates internal states and functions and can be configured and extended. Each layer of S3 is comprised of multiple ABBs, which collaborate to carry expected activities. Considering the adaptive feature of business scenarios, each project may have to configure and customize the layered ABB template library for proprietary requirements and constraints. In addition, the ABB templates imply certain relationships between some ABBs.

This paper reports our research aiming to extend and build the variation analysis [7]-oriented formalism of a modeling environment, with flexibility and extensibility, to enable and support solution-level system design and implementation based on architectural thinking. Specially, we utilize the power of formal methods [8] to model solution-level architectural artifacts and their relationships, whose formalization enables event-based variation notification and propagation analysis. Note that variation here refers to different configuration and customization from architectural artifact template library. Such variation may be propagated and lead to changes in related architectural artifacts.

The solution-level design in the paper refers to metadata-level design of architectural building blocks; and the presented modeling approach is applicable to any layered or tiered architecture. Note the slight differences between several words used throughout the paper, component, architectural artifact, and ABB. A component refers to a comprising part of the structure of an SOA solution with a relatively clearly defined structural boundary. An ABB refers to a highly reusable component in an SOA solution architecture. An architectural artifact

978-0-7695-3840-2/09 $26.00 © 2009 IEEE
DOI 10.1109/CLOUD.2009.86

is a more generic term referring to either a component or an ABB from architecture perspective.

The remainder of this paper is organized as follows. In Section 2, we discuss related work. In Section 3, we introduce architectural-centric modeling notations, variation propagation modeling, and solution modeling. In Section 4, we explain our implementation of ABB modeling by extending the UML modeling technique. In Section 5, we introduce our prototype system and explain how it validates a customized SOA solution, as well as an overview of our modeling process. In Section 6, we make conclusions.

2. Related work

Some researchers have studied variation analysis in Web services development [9-11]. Variation is well known as variability management in the generic software development process [12]. Svahnberg et al. define software variability as the ability of a software system to be efficiently extended, changed, customized or configured for use in a particular context [13]. Software variability is usually classified into different categories, from different perspectives such as variation points, variation realization techniques, and feature variations. A variation point refers to a design decision that is consciously left open for software engineers to decide based on some specific situations.

Jiang et al. [9] identify a set of variation points from three resources: WSDL documents, service endpoints, and business logics. However, their modeling stays at simple application examples. The relationships between variation points are not examined.

Kim and Doh [10] identify three tiers (presentation tier, service tier, and application tier) in a Web service design and use UML to model variation points in each tier and their relationships. However, their modeling currently stays at a rudimentary stage and has not been used in any practice.

Ruokonen et al. [11] identify a category of variation needs and types relevant for SOA-based systems, based on two dimensions, system concepts and development process. However, their variation definitions stay at a high level; thus, they do not provide normative guidance for software architects.

Mezini and Ostermann propose a feature-oriented and aspect-oriented modularization approach to manage variability in the context of system families [14].

Various types of formal methods [8] such as Model-Driven Development (MDD) approaches have been widely used to increase software design and development productivity and reduce time-to-market in a systematic manner [15-17]. Representative examples include OMG's MDA [18], Domain Specific Modeling (DSM) [19], and Software Factories [20] from Microsoft.

Mattsson et al. find out that traditional MDD cannot automate the enforcement of architecture on detailed design due to its inability to model architectural design rules [21]. They emphasize the importance of formally modeling architectural design rules. Our reported work in this paper forms a foundation to model architectural artifacts, relationships, and layered verification rules.

Our ABB templates [2, 4-6] provide a comprehensive set of architectural building blocks for each of the nine layers in the S3 model based on industry best practices. The templates can be used as a starting point for software architects to quickly configure and design a prototype for a specific SOA solution. However, although the templates require some relationships between comprising architectural artifacts, these relationships are implicit and require significant learning curves and personal experiences. Therefore, this research aims to build a formalism-based model driven design environment to guide software architects in constructing an appropriate SOA solution.

3. Foundations for ABB modeling

To ease our discussion, Figure 1 shows the recommended ABB configuration for the Service Consumer layer of S3. Eight ABBs are identified: A *consumer* ABB represents an external user. A *presentation (view)* ABB is responsible for communicating to and from external consumers. A *presentation controller* ABB is the coordinator for all other ABBs in the layer. A *consumer profile* ABB is responsible for getting customer-specific information. An *access control* ABB provides authentication/authorization capabilities. A *format transformation* ABB is responsible for translation of query content format. A *configuration rule* ABB is responsible for hosting rules that define how the ABBs can be configured. A *cache* ABB is responsible for temporarily storing consumer interaction-related data. Detailed ABB definition and configuration can refer to our previous report [6].

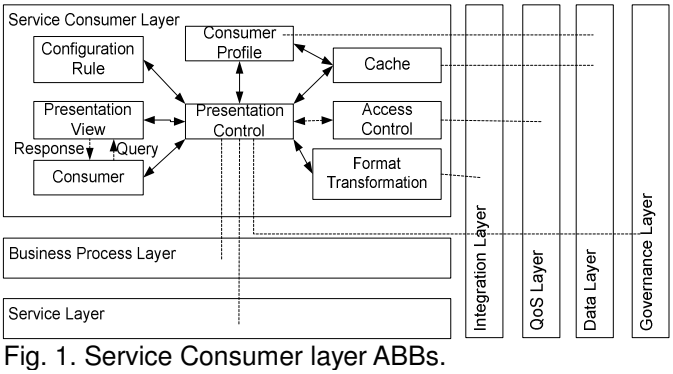

Fig. 1. Service Consumer layer ABBs.

3.1 ABB modeling

To facilitate our formalization, Table 1 summarizes the fundamental notations and their semantic meanings, which will be discussed in detail in the following sections and used throughout the paper.

TABLE 1
NOTATIONS OF ABB MODELING

Notations	Semantic Meanings
X	a layer in S3
Φ_X	ABB type set in layer X
Ω_X	ABB instance set in layer X
A_{ABB}^X	ABB type A at layer X
$a^{ABB,A}$	an instance of ABB type A (i.e., with stereotype A)
$*_A^X$	# of instances of A_{ABB}^X
$\mathrm{M}_m^n\left\langle A_{ABB}^X, B_{ABB}^Y \right\rangle$	ABB type A and ABB type B has an m:n mapping relationship
$\mathrm{M}_p^q\left\langle a^{ABB,A}, b^{ABB,B} \right\rangle$	an instance of ABB type A and an instance of ABB type B has a p:q mapping relationship
M_p^q	a p:q mapping relationship between either a pair of ABB types or a pair of ABB instances
$\subseteq$	compliant relationship between ABB instances with that between corresponding ABB types (both mapping and propagation)
$\mathrm{P}_{\mathrm{M}_p^q}^{R_{\mapsto}^t}\left(a^{ABB,A}, b^{ABB,B}\right)$	protocol between a pair of ABB instances (M_p^q for mapping and $R_{\mapsto}^t$ for propagation)
$\mathrm{P}_{\mathrm{M}_m^n}^{R_{UML}}\left(A_{ABB}^X, B_{ABB}^Y\right)$	protocol between a pair of ABB types (M_m^n for mapping and R_{UML} relationship in UML 2.0)
S	an SOA solution
$a^{ABB,A} \mapsto b^{ABB,B}$	a variation publisher/subscriber relationship exists from $a^{ABB,A}$ to $b^{ABB,B}$
$R_{\mapsto}^t$	type *t* publisher/subscriber relationship
$\prec$	Compliant protocol between a pair of ABB instances with their corresponding ABB types

Definition 1. An *Architectural Building Block (ABB)* is a component in S3 that encapsulates internal states and functions and can be configured and extended. An *ABB type* refers to an ABB we identified representing a typical fine-grained building class in an SOA solution. An *ABB instance* refers to a building block in a specific SOA solution implementing an ABB type. An *ABB component* refers to either an ABB type or an ABB instance.

Definition 2. A *mapping relationship* refers to the cardinality between two ABB components, if there is relationship between them. A mapping relationship can be represented by a pair of sets m:n, each falling into one of the three possibilities: {1}, {0..*}, or {1..*}.

$$\mathrm{M}_m^n \Rightarrow \mathrm{M}_m^n \in \left\{null, \mathrm{M}_1^1, \mathrm{M}_1^{0..*}, \mathrm{M}_1^{1..*}, \mathrm{M}_{0..*}^1, ...\right\} \quad \text{(R1)}$$

The cardinality definitions can be used to detect and verify component-level variations. For the same reason, we define a *null* relationship to imply that no relationship exists between two ABB components.

Definition 3. A *compliant mapping relationship* requires that the mapping relationship between a pair of ABB instances be either the same as or stronger than the one defined between their corresponding ABB types:

$$\mathrm{M}_p^q\left\langle a^{ABB,A}, b^{ABB,B} \right\rangle \subseteq \mathrm{M}_m^n\left\langle A_{ABB}^X, B_{ABB}^Y \right\rangle \Rightarrow$$
$$\left((q \subseteq n) \wedge (p \subseteq m)\right) \vee \left(\mathrm{M}_p^q = null \wedge \mathrm{M}_m^n = null\right) \quad \text{(R2)}$$

3.2 Variation propagation modeling

3.2.1. Propagation relationship

To maintain loose coupling between ABBs in an SOA solution, we applied the publisher/subscriber design pattern [22] to manage variation propagation and synchronize cooperative ABBs. By notifying all of its subscribers about changes, a publisher enables single-directional change propagation.

Definition 4. For each identified single-directional relationship between a pair of ABB instances, the ABB instance that is independent from the other one is assigned as a variation publisher; the ABB instance that depends on the other one is assigned as a variation subscriber. Their relationship can be represented by: $a^{ABB,A} \mapsto b^{ABB,B}$, where: $a^{ABB,A}$ and $b^{ABB,B}$ denote two ABB instances, with the former denoting the variation publisher in the relationship and the latter denoting the variation subscriber.

The propagation relationships between each pair of ABBs can be normalized as one or two single-directional relationships (a bi-lateral relationship can be broken into two single-lateral relationships). Such a relationship is enabled by the variation subscriber registering itself at the variation publisher.

3.2.2. Propagation between ABB types

Properly designed propagation relationships between ABB types can be used for two purposes. First, since metamodel is typically maintained by the highest-level architects, the defined relationships can be used as guidance for actual model design. Second, defined relationships can be used to validate and verify actual propagation relationships between ABB instances.

Propagation relationships can be overwritten at ABB instances from ABB types within certain degrees if necessary. For example, a two-dimensional association relationship can be strengthened into one-dimensional directed association. We summarize our considerations of propagation relationships between ABB types and ABB

instances in Figure 2.

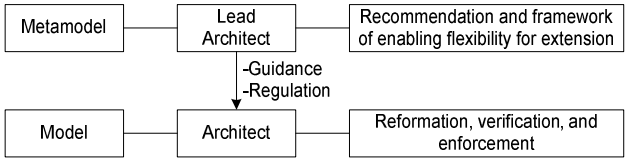

Fig. 2. Propagation relationship specification model.

As shown in Figure 2, metamodels are typically maintained by lead architects and models are constructed by individual SOA architects. Our strategy is that metamodels can leverage and extend normal UML 2.0 relationships. The categorized propagation relationships summarized in Table 2 can be recommended but do not have to be enforced. On the other hand, models should be allowed to use the propagation relationships in Table 2. While architects could still try to use UML 2.0 relationships, some reformation work will be performed underlying (e.g., break a two-dimensional association into two one-dimensional associations), and incompatible relationships (e.g., package import) will be warned and enforced to be removed from a valid model.

TABLE 2.
ABB-LEVEL VARIATION MANAGEMENT.

Cat	Covered UML relationships	Descriptions	Variation management
1	directed association, dependency, usage	Customized dependency relationship	Registration at the depended ABB publisher
2	aggregation, composition	Whole/part relationship	Registration at the whole ABB (publisher)
3	implements, realization	Abstraction/implementation relationship	Registration at the abstraction ABB publisher)
4	generalization	Super-/sub-class relationship	Registration at the sub-class ABB publisher

Definition 5. A compliant propagation relationship refers to the propagation relationship defined between a pair of ABB instances that must be compliant with that defined between the corresponding pair of ABB types. The requirement can be represented as R3:

$$\forall\left(R^t_{\mapsto}\left\langle a^{ABB,A}, b^{ABB,B}\right\rangle\right)\Rightarrow$$
$$R^t_{\mapsto}\left\langle a^{ABB,A}, b^{ABB,B}\right\rangle \subseteq R_{UML}\left\langle A^X_{ABB}, B^Y_{ABB}\right\rangle \quad \text{(R3)}$$

As mentioned earlier in this paper, we considered simplified publisher/subscriber relationship. Thus, R3 can be simplified into R4:

$$\forall\left(R_{\mapsto}\left\langle a^{ABB,A}, b^{ABB,B}\right\rangle\right)\Rightarrow$$
$$R_{\mapsto}\left\langle a^{ABB,A}, b^{ABB,B}\right\rangle \subseteq R_{UML}\left\langle A^X_{ABB}, B^Y_{ABB}\right\rangle \quad \text{(R4)}$$

3.3 SOA solution definition

In this section, we discuss how to model an SOA solution based on ABB propagation modeling.

3.3.1. Propagation path

When an ABB-related variation occurs at runtime, it needs to be propagated to all related ABB instances.

Definition 6. A propagation set depicts an ordered set of all either directly or indirectly impacted ABB instances in an SOA solution when a variation happens at one contained ABB instance. A propagation set can be denoted by a directed graph represented by a 4-tuple $G^P_a = (PN, PA, a, PSet)$, where PN represents a set of ABB instances; PA represents a set of directed arcs; a represents the starting point of ABB instance; and $PSet$ represents a set that includes all propagation paths starting from a. Each arc links two nodes if there is a propagation relationship between the two ABB instances. A propagation path of the ABB instance a depicts a subset of its propagation set, starting from a and cannot further propagate. A propagation path of an ABB instance a can be denoted by a directed graph $path = (\vec{N}, \vec{A}, a)$,

where: $\vec{N} = \{a, N_1, N_2, ..., Nn\}$,

$\vec{A} = \left\{R^t_{\mapsto}(a, N_1), R^t_{\mapsto}(N_1, N_2), R^t_{\mapsto}(N_2, N_3), ...R^t_{\mapsto}(N_{n-1}, N_n)\right\}$,

$\vec{N} \subseteq PN, \vec{A} \subseteq PA$

Definition 5 implies that:

$\forall \alpha \in PA \Rightarrow$

$$\exists\left(\left(a^{ABB,A} \in N\right) \wedge \left(b^{ABB,B} \in N\right) \wedge \left(R^t_{\mapsto}\left\langle a^{ABB,A}, b^{ABB,B}\right\rangle\right)\right) \quad \text{(R5)}$$

and

$PSet = \{P_1, P_2, ..., P_k\}, P_i = \{a, N_1, N_2, ..., N_n\},$

$i \in [1, k], N_j \in PN, j \in [1, n], a \mapsto N_1, N_1 \mapsto N_2, ..., N_{n-1} \mapsto N_n$

A propagation path depicts a possible farthest propagation pathway starting from an ABB instance. A propagation points out a message passing path between involved ABB instances at runtime.

3.3.2. ABB protocol

Putting together mapping relationship and propagation relationship, we can define protocols between ABB types and ABB instances.

Definition 8. One protocol between a pair of ABB types contains two elements: a mapping relationship (as defined in Definition 1) and a propagation relationship referring to a UML 2.0 relationship that defines the correlation between two ABB types. Such a protocol can be represented as follows:

$\mathrm{P}_{\mathrm{M}_m^n}^{R_{UML}}\left(A_{ABB}^X, B_{ABB}^Y\right)$, where:

A_{ABB}^X and B_{ABB}^Y denote two ABB types; M_m^n denotes that the cardinality between the two ABB types is $M_m^n\left(A_{ABB}^X, B_{ABB}^Y\right)$; there is a correlation relationship between the two ABB types that can be represented by a UML 2.0-defined relationship R_{UML}.

Definition 9. One protocol between a pair of ABB instances contains two elements: a mapping relationship (as defined in Definition 1) and a propagation relationship referring to a publisher/subscriber relationship that defines the direction of variation propagation. Such a protocol can be represented as follows:

$\mathrm{P}_{\mathrm{M}_p^q}^{R_{\mapsto}^t}\left(a^{ABB,A}, b^{ABB,B}\right)$, where:

$a^{ABB,A}$ and $b^{ABB,B}$ denote two ABB instances; M_p^q denotes that the cardinality between the two ABB instances is $M_p^q\left(a^{ABB,A}, b^{ABB,B}\right)$; there is a publisher/subscriber relationship between the two instances $a^{ABB,A} \mapsto b^{ABB,B}$, where $a^{ABB,A}$ is the publisher and $b^{ABB,B}$ is the subscriber; and the publisher/subscriber type $R_{\mapsto}^t$ between the two ABB instances is of type t $R_{\mapsto}^t$ (In this paper since we only consider one type of generic publisher/subscriber relationship, it can be simplified as $R_{\mapsto}$).

Definition 10. A compliant ABB protocol refers to the protocol defined between a pair of ABB instances must be compliant with that defined between the corresponding pair of ABB types in two aspects: one is their mapping relationship must be compliant; the other one is their propagation relationship must be compliant.

$$\mathrm{P}_{\mathrm{M}_m^n}^{R_{UML}}\left(A_{ABB}^X, B_{ABB}^Y\right) \prec \mathrm{P}_{\mathrm{M}_p^q}^{R_{\mapsto}^t}\left(a^{ABB,A}, b^{ABB,B}\right) \Rightarrow \left(R_{\mapsto}^t \subseteq R_{UML}\right) \wedge \left(\mathrm{M}_p^q \subseteq \mathrm{M}_m^n\right) \quad \text{(R6)}$$

3.3.3. Definition of an SOA Solution

Definition 11. An SOA solution S is defined as a tuple as follows: $S = \langle \Phi, \Omega \rangle$, where:

- Φ represents the metamodel of the solution, which is in turn a tuple: $\Phi = \langle \Phi_\Phi, \Phi_P \rangle$, where: Φ_Φ represents nine layers of ABB types: $\Phi_\Phi = \bigcup_{i=1}^{9} \Phi_{X_i}$, Φ_P represents the relationships between the defined ABB types:
- Ω represents the actual model of the solution, which is in turn a tuple: $\Omega = \langle \Omega_\Omega, \Omega_P \rangle$, where: Ω_Ω represents nine layers of ABB instances: $\Omega_\Omega = \bigcup_{i=1}^{9} \Omega_{X_i}$, Ω_P represents the relationships between the defined ABB instances.

4. SOA solution modeling

Based on the presented formal ABB modeling, we now discuss how we leveraged the model-driven approach to extend UML to implement ABB-based SOA solution modeling as an example. UML has become a widely accepted industry standard for software system modeling [23]. The major technical challenge we encountered is how to implement our formal ABB representation using UML modeling. Particularly, we had to tackle two issues: how to model variation propagation between ABB instances and how to model ABB-based SOA solutions.

4.1. Propagation between ABB instances

UML provides a comprehensive set of relationship types to model relationships between entities (e.g., class, interface, component, and package) [9]. In this paper, we are only interested in the relationship types that may lead to variation propagation through ABB instances. For example, consider a *Presentation Controller* ABB that *uses* an *Access Control* ABB. If an access control method name in the *Access Control* ABB is changed, the *Presentation Controller* ABB should be informed to make consequent changes.

We examined all relationship types defined in UML 2.0. Some relationship types will not cause component-level variation propagation (e.g., substitution), or have equivalents (e.g., abstraction), or are irrelevant to the semantics of ABB instances (e.g., instantiation and binding). At ABB level, these relationships will not be allowed so we remove them from our consideration. After this process, we obtain a set of 10 relationships: association, link, directed association, aggregation, composition, generalization, implements, realization, dependency, and usage.

Among the remaining set, some relationship types are one-directional (e.g., directed association and aggregation) while others are bi-directional (association and link). Because we intend to analyze variation propagation, direction is important here. Therefore, for each bi-directional relationship, we break them into two single-directional ones.

Meanwhile, considering component-level variation propagation, some relationships will cause the same or similar action. We organize the studied relationships into four categories as shown in Table 2. Detailed definition

for each UML relationship can be found from UML 2.0 specification [23] and will be followed in our study.

The major reason why we categorize the relationships between ABBs is for enabling variation propagation analysis and management, which is also the criterion of how we categorize the relationships. Category 1 represents customized dependency relationship between two ABBs, which covers three types of UML relationships: *directed association*, *dependency*, and *usage*. Category 2 represents whole/part relationship between two ABBs, which covers two types of UML relationships: *aggregation* and *composition*. Category 3 represents abstraction/implementation relationship between two ABBs, which covers two types of UML relationships: *implements* and *realization*. Category 4 represents super-/sub-class relationship between two ABBs, which covers the UML relationship *generalization*.

Each category implies a proprietary variation management approach. Table 2 provides the guidance on how to apply the publisher/subscriber pattern and assign variation publisher/subscriber roles to each type of ABB relationship. The strategy is that the ABB that will be notified for variation propagation will be assigned as a variation subscriber; and the other ABB initiating a variation will be assigned as a variation publisher. For Category 1, each covered UML relationship implies a dependency relationship between the two ABBs. Therefore, the depended ABB is assigned as the variation publisher and the dependent ABB is assigned as the variation subscriber; the dependent ABB registers itself at the depended ABB. For Category 2, the ABB as a part is assigned as the variation subscriber and the ABB as the whole is assigned as the variation publisher; the part ABB registers itself at the whole ABB. For Category 3, the ABB as the abstraction is assigned as the variation publisher and the ABB as the implementation is assigned as the variation subscriber; the implementation ABB registers itself at the abstraction ABB. For Category 4, the ABB as the sub-class is assigned as the variation publisher and the ABB as the super-class is assigned as the variation subscriber; the super-class ABB registers itself at the sub-class ABB.

Six levels of variations should be propagated: data entity, message, business primitive, business construct, business protocol, and business process [2]. All of these variations can be propagated through the same publisher/subscriber approach. However, it is useful to differentiate between levels of variations through propagation, so that proper action can be conducted effectively and efficiently. For example, if a variation happened at a variation publisher is at an operation level, its variation subscriber may only update its corresponding operation invocation. If the propagation only informs a variation occurrence without variation level, though, the variation subscriber may not be able to take right action promptly. This simple example illustrates the necessity of differentiating variation levels. To address this need, our solution is to extend the original publisher/subscriber pattern by associating variation level information. When a publisher/subscriber relationship is set up between two ABB instances, one of the six levels of variation will be bound to the publisher/subscriber relationship. To simplify the studied domain, without losing generality, in this paper we only consider the original default publisher/subscriber format. In other words, when any level of variation happens at a variation publisher, its corresponding variation subscribers will be informed in the same way. Studying variation propagation at various levels in detail will be our future research.

4.2. ABB-based SOA solution modeling

In this section, we introduce how we extend UML to model ABBs associated with variations in an SOA solution. Starting from UML 2.0, metamodels specify the abstract syntax and semantics of UML modeling concepts. On top of metamodel, profiles and constraints provide mechanisms to extend the existing UML metamodel to model specific applications. In more detail, the UML metamodel defines the basic stereotypes (i.e., data types) that users can utilize to build UML models. Examples of basic stereotypes are class, package, and association. A user-defined metamodel defines newly added stereotypes and their relationships. A profile carries stereotypes and constraints can be applied by users to create instances of stereotypes in UML models. In other words, a user-defined profile defines application-specific building blocks that users can use to build specific UML models.

In this research, we extend the UML metamodel to model ABB-based SOA solutions. Each ABB is modeled as a UML stereotype; stereotypes representing all ABBs defined in one layer are grouped in one package as a profile. The rationale is that, every building block in an SOA solution is an instance of an identified ABB. In other words, an ABB can be considered as a class with specific definitions that can be instantiated into final building blocks.

As S3 illustrates, a typical SOA solution can be organized into a nine-layer structure. The final code structure can consequently be organized into nine packages. Note that these nine packages only represent the top-level software structure; more packages can be identified if needed. Therefore, we establish a mapping between SOA solutions with UML models, by mapping an S3 layer to a UML package and mapping an ABB to a UML stereotype.

4.3. Metamodel-centered adaptive modeling

According to our extensions to UML to model SOA solutions using ABBs, we created three kinds of artifacts:

metamodel, profile, and palette. For each layer of the S3 model, we first created a metamodel representing all identified ABBs as stereotypes as the conceptual model of the layer, then we created a package of profiles representing each ABB as a stereotype, and then we created a palette containing all ABBs as building blocks for users to use to create UML models.

We found that there are inherent relationships among the three concepts: metamodel, profile, and palette. As a matter of fact, all of the three concepts define stereotypes that can be used by users to build UML models. A palette is a typical way for a UML modeling tool to prepare a set of available stereotypes for users to pick from and draw UML diagrams. A profile carries defined stereotypes for users to apply to models. A metamodel is the source to formally define stereotypes.

Due to the intrinsic relationships among the three concepts, if we create them separately, whenever we make changes to any ABBs, we have to apply the same changes to all three types. This approach of manual consistency control is not only inefficient but also error-prone.

Our solution is to utilize the inherent relationships among the three concepts and use the metamodel as the basis to realize automatic synchronization among them. The details of the idea can be summarized as in the pseudo code shown below. We first construct the metamodel for each layer. Then for each metamodel, we identify the stereotypes and create a corresponding profile containing all ABBs as stereotypes. Afterwards we create a corresponding palette comprising all ABBs as building blocks. Any UML modeling environment typically already contains a built-in palette representing basic building blocks in hierarchies for UML modeling, such as class, association, class diagram, and so on. To differentiate between our SOA solution-oriented palette and embedded-in palette, we organize palette in model libraries instead.

Alg. 1: Metamodel-centered SOA solution model management

1. If they do not exist, create all three categories of elements.
   1.1 For each of the nine layers of the S3 model,
   Create a metamodel with all identified ABBs;
   1.2. For each of the nine metamodel,
   1.2.1. Create a profile;
   1.2.2. Iterate through all ABBs in the metamodel,
   Create a stereotype in the profile;
   1.3. For each of the nine metamodel,
   1.3.1. Create a model library;
   1.3.2. Iterate through all ABBs in the metamodel,
   Create a building block in the model library;
2. If already existed but a change is required,
   2.1 make changes to the corresponding metamodel;
   2.2 re-generate the corresponding profile elements only;
   2.3 re-generate corresponding model library elements.

Whenever changes are required, we will modify the corresponding metamodels. Afterwards, by re-generating corresponding profile elements and model library elements, all three types of model elements can be synchronized. It should be noted that intelligence is necessary here to ensure that only involved profile elements and model library elements be re-generated, so that other elements with customized code (e.g., with user-added derived stereotypes) can stay untouched.

4.4. Usages of metamodel for generating solution patterns

The industry best practice we use not only leads to the identification of ABBs, but also depicts the relationships between the ABBs. As shown in Figure 3, since a metamodel is a class diagram, these relationships can be represented in the metamodel using standard UML relationship elements. For example, there is a one-to-many relationship between a "Presentation Controller" stereotype and a "Presentation" stereotype. These relationships can help to build SOA solutions using instances of ABBs (i.e., stereotypes). Figure 3 shows an example structure of a Services Consumer layer of an SOA solution. As shown in Figure 3, each of the ABB identified has an instance; these ABB instances are interconnected according to predefined relationships.

5. SOA solution validation

In this section, we discuss how to validate a customized SOA model. We organize architectural rules in a three-level hierarchy: (1) system-level rules, (2) layer-level rules, and (3) ABB-level rules. A set of top-level rules is established at global S3 model-wide for any SOA solution. These rules are to be enforced with the highest priority over any other rules. For example, each ABB type can only belong to one layer. Each layer can define a specific set of rules as layer-level rules, which have to be enforced with the second highest priority. ABB-level rules refer to the relationships defined between ABB types that have to be enforced on corresponding ABB instances.

5.1. System-level rules

Definition 12. The system-level rule set for an SOA solution $S = \langle \Phi, \Omega \rangle$ contains six predefined rules:

1. $\forall A_{ABB}^{X} \Rightarrow \exists(\Phi_X \in \Phi_\Phi) \wedge \left(A_{ABB}^{X} \in \Phi_X\right) \wedge \left(\exists A_{ABB}^{X} \wedge \exists A_{ABB}^{Y} \Rightarrow X = Y\right)$
2. $\forall a^{ABB,A} \Rightarrow \exists(\Omega_X \in \Omega_\Omega) \wedge \left(a^{ABB,A} \in \Omega_X\right) \wedge \left(\exists a^{ABB,A} \in \Omega_X \wedge a^{ABB,A} \in \Omega_Y \Rightarrow X = Y\right)$
3. $\Phi_X = null \Rightarrow \Phi_X = \phi$

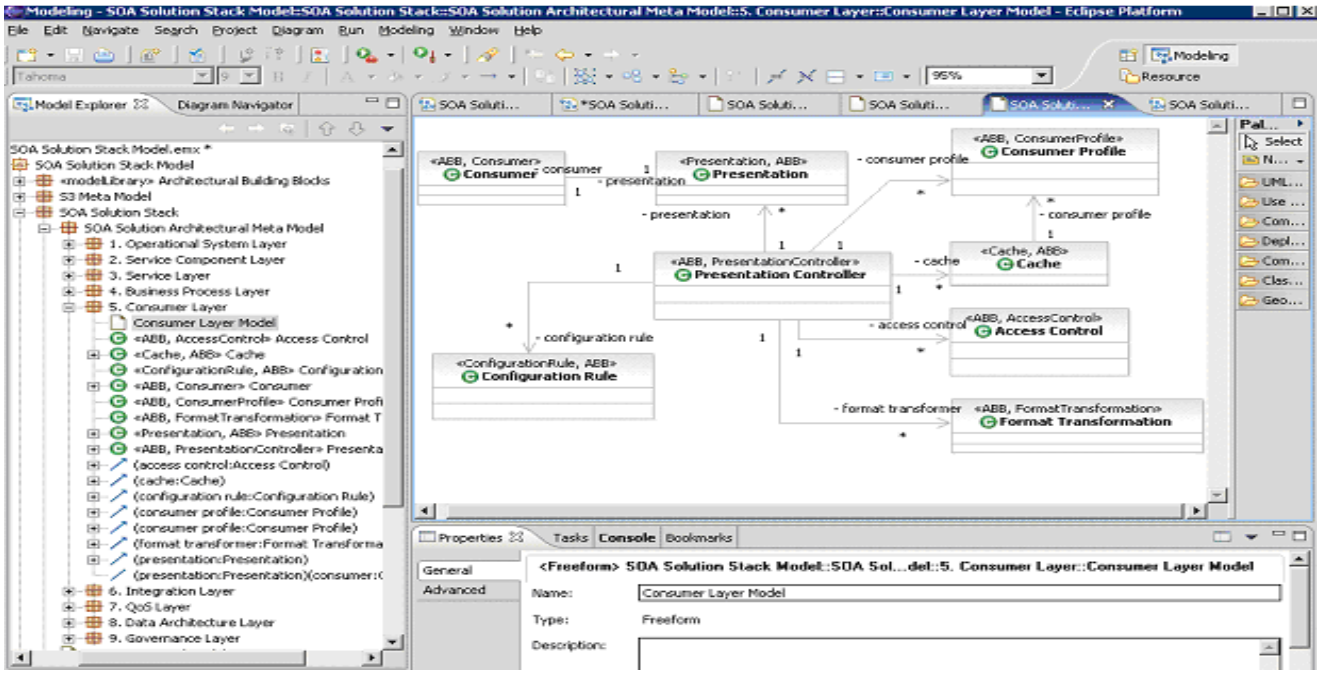

Fig. 3. Generated default modeling template.

4. $\Phi_X = \phi \Rightarrow \Omega_X = \phi$

5. $*_A^X \geq 0$

$$\forall\left(\mathrm{P}_{\mathrm{M}_m^n}^{R_{UML}}\left\langle A_{ABB}^X, B_{ABB}^Y\right\rangle \wedge \mathrm{P}_{\mathrm{M}_p^q}^{R_{\mapsto}^t}\left\langle a^{ABB,A}, b^{ABB,B}\right\rangle\right) \Rightarrow$$

6.

$$\mathrm{P}_{\mathrm{M}_m^n}^{R_{UML}}\left(A_{ABB}^X, B_{ABB}^Y\right) \prec \mathrm{P}_{\mathrm{M}_p^q}^{R_{\mapsto}^t}\left(a^{ABB,A}, b^{ABB,B}\right)$$

Rule 1 denotes that each ABB type A_{ABB}^X belongs to and only belongs to one specific layer X. Rule 2 denotes that each instance of ABB type A belongs to and only belongs to one layer X to which the ABB type A belongs. Rule 3 denotes that for a particular SOA solution, one layer may contain no ABBs, meaning that the layer is not necessary for the solution. Rule 4 denotes that if a specific layer does not exist in the metamodel, the layer does not exist in the actual SOA model. Rule 5 denotes that one ABB type A may or may not have instances in a specific SOA solution, and one ABB type A may have more than one instance in a specific SOA solution. Rule 6 denotes that a protocol between a pair of ABB instances in a specific SOA solution is required to be compliant with that defined between the corresponding ABB types in the corresponding metamodel of the SOA solution. In general, the top-level rules are implicitly defined and cannot be modified throughout the modeling of any SOA solution.

5.2. Layer-level rules

Each layer in the S3 model can carry a specific set of rules as layer-level rules. Taking the Service Consumer layer as an example, its layer-level rule set possesses one rule R6:

$$\Omega_{Consumer} \neq \phi \Rightarrow$$

$$iff\left(\begin{array}{l} presentation\_controller^{ABB,\mathrm{Pr}esentation\_Controller} \\ \in \Omega_{Consumer\_Layer} \wedge *_{presentation\_controller}^{Consumer\_Layer} = 1 \end{array}\right) \quad (R6)$$

R6 denotes that if the Service Consumer layer model is not empty, then there has to be one and only one instance of *Presentation Controller* ABB exists. This rule can be enforced by setting the multiplicity of the *Presentation Controller* ABB type to be "1."

Each layer may carry some predefined layer-level rules. Typically, these layer-level rules can be modified by chief architects at the time of construction of a metamodel for a specific SOA solution. For example, a comprehensive application may allow multiple instances of *Presentation Controller* ABB exist in its Service Consumer layer. This rule can be represented as R7 and can be set up by setting the multiplicity of the *Presentation Controller* ABB type to be "1..*".

$$\Omega_{Consumer} \neq \phi \Rightarrow$$

$$iff\left(\begin{array}{l} presentation\_controller^{ABB,\mathrm{Pr}esentation\_Controller} \\ \in \Omega_{Consumer\_Layer} \wedge *_{presentation\_controller}^{Consumer\_Layer} \geq 1 \end{array}\right) (R7)$$

Meanwhile, application-specific layer-level rules can be added if necessary. For example, one particular PDA-oriented application may require that at least one pair of

(PDA_Consumer, PDA_Presentation) instances are created for the ABB type pair (Consumer, Presentation), meaning that the solution has to support PDA users by providing corresponding PDA-specific interface presentation generation mechanisms. The rule can be represented as follows:

$$\Omega_{\mathrm{Pr}esentation\_Layer} \wedge *^{\mathrm{Pr}esentation\_Layer}_{Consumer} \geq 1 \qquad \text{(R8)}$$

5.3. ABB-level rules

ABB-level rules define the mutual connections between ABB types that need to be enforced on corresponding ABB instances. The connection relationship in our concern includes cardinality relationship and propagation relationship. Such relationships can be deduced and transformed into rules to regulate the construction of SOA models using ABB instances. For example, a chief architect may define that there is a 0:m dependency relationship between a *Presentation Controller* ABB type and an *Access Controller* ABB type in the Service Consumer layer, which can be represented as follows:

$$\mathrm{P}^{R_{Dependency}}_{\mathrm{M}_1^{0..*}} \left(\begin{array}{l} \mathrm{Pr}esentation\_Controller^{Consumer\_Layer}_{ABB}, \\ Access\_Controller^{Consumer\_Layer}_{ABB} \end{array} \right) \text{(R9)}$$

The above rule can be deduced into the following set of rules comprising three rules regarding related ABB instances:

$$(1)\ \mathrm{P}^{R_{\mapsto}}_{\mathrm{M}_1^{0..*}} \left(\begin{array}{l} presentation\_controller^{ABB,Consumer\_Layer}, \\ access\_controller^{ABB,Consumer\_Layer} \end{array} \right)$$

$$(2)\ \mathrm{P}^{R_{\mapsto}}_{\mathrm{M}_1^{1..*}} \left(\begin{array}{l} presentation\_controller^{ABB,Consumer\_Layer}, \\ access\_controller^{ABB,Consumer\_Layer} \end{array} \right)$$

$$(3)\ \mathrm{P}^{R_{\mapsto}}_{\mathrm{M}_1^{1}} \left(\begin{array}{l} presentation\_controller^{ABB,Consumer\_Layer}, \\ access\_controller^{ABB,Consumer\_Layer} \end{array} \right)$$

The deducted rules indicate acceptable relationships regarding corresponding ABB instances in an SOA model. In other words, if an SOA model contains a pair of ABB instances *Presentation Controller* and *Access Controller*, their relationship has to be one of the above three possibilities to be considered as a valid relationship.

Using the same method, each defined relationship between a pair of contained ABB types in a metamodel can be represented as an ABB-level rule; each such rule can be deducted into a set of rules for a corresponding pair of ABB instances. Accumulating them all, for each metamodel for an S3 layer, we can obtain a knowledge base containing a set of relationship rules for acceptable relationships for the S3 layer.

In summary, each relationship defined between a pair of ABB types in a metamodel can be deducted into a set of rules for ABB instances. Each defined relationships between a pair of ABB instances can be transformed into a normalized rule to be compared with the deducted rule sets. If it can be found in the knowledge base, it is a valid relationship; otherwise, it should be rejected.

5.4. Modeling environment and process

With the establishment of ABB-based modeling technique and solution validation mechanism, we constructed SOA Modeling Environment (SOA-ME) as a prototype modeling tool. It is built upon IBM Rational Software Architect (RSA) as its development platform to exploit the comprehensive architectural design artifacts and interfaces of the RSA.

SOA-ME can be logically divided into three sections: rule generation, static validation, and dynamic validation. The rule generation section creates system-level rules, layer-level rules, and ABB-level rules into the *rule knowledge base*. The static validation section takes the input of a customized SOA model, analyzes its contents, while checking against the rule knowledge base. The result of the static model parser is a valid SOA model, which can be deployed for runtime usages. The dynamic validation section examines the messages passed between ABB instances and re-engineers the propagation paths. Then it compares the generated sets with the stored propagation paths created at the static validation phase. If the two sets are equivalent, the validation result is positive.

6. Conclusions

In this paper, we present our design and development of a systematic SOA solution modeling and variation propagation analysis and ABB template library. The architectural artifacts and variation propagation modeling method enables software architects to manage SOA solution-level quality. Our research lays a foundation to build a practical engineering tool to guide software architects in designing SOA solutions toward application cloud services. The formalization of variation-oriented analysis for architectural artifacts allows solution-level enforcement and performance analysis, and supports architectural evolutionary changes.

For our future study, we plan to explore rule-based formal variation propagation analysis and verification algorithms in the context of our modeling tool. In addition to compile-time static verification, we plan to study dynamic variation analysis to provide solution adaptability for run-time evolutionary changes in service provisioning scenarios for Cloud Computing. It is noted that the presented modeling approach can be applied to any layered or tiered architecture.

7. References

[1] L.-J. Zhang and Q. Zhou, "CCOA: Cloud Computing Open Architecture", in Proceedings of *IEEE International Conference on Web Services (ICWS)*, 2009, Los Angeles, CA, USA, pp.

[2] L.-J. Zhang, J. Zhang, and H. Cai, *Services Computing*. Springer, 2007.

[3] A. Arsanjani, L.-J. Zhang, M. Ellis, A. Allam, and K. Channabasavaiah, "S3: A Service-Oriented Reference Architecture", *IT Professional*, May, 2007: pp. 10-17.

[4] L.-J. Zhang and J. Zhang, "Design of Service Component Layer in SOA Reference Architecture", in Proceedings of *33rd Annual IEEE International Computer Software and Applications Conference (COMPSAC)*, Jul. 20-24, 2009, Seattle, WA, USA, pp. 474-497.

[5] L.-J. Zhang and J. Zhang, "Componentization of Business Process Layer in The SOA Reference Architecture", in Proceedings of *IEEE International Conference on Services Computing (SCC)*, 2009, Bangalore, India.

[6] L.-J. Zhang, J. Zhang, and A. Allam, "A Method and Case Study of Designing Presentation Module in an SOA-based Solution Using Configurable Architectural Building Blocks (ABBs)", in Proceedings of *IEEE International Conference on Services Computing (SCC)*, Jul. 8-11, 2008, Honolulu, HI, USA, pp. 459-467.

[7] L.-J. Zhang, A. Arsanjani, A. Allam, D. Lu, and Y.-M. Chee, "Variation-Oriented Analysis for SOA Solution Design", in Proceedings of *IEEE International Conference on Services Computing (SCC)*, Jul. 9-13, 2007, Salt Lake City, UT, USA, pp. 560-568.

[8] I. Traore and D.B. Aredo, "Enhancing Structured Review with Model-Based Verification", *IEEE Transactions on Software Engineering*, 2004, 30(11): pp. 736-753.

[9] J. Jiang, A. Ruokonen, and T. Systa, "Pattern-Based Variability Management in Web Service Development", in Proceedings of *Third IEEE European Conference on Web Services (ECOWS)*, Nov. 14-16, 2005, Växj, Sweden, pp. 83-94.

[10] Y. Kim and K.-G. Doh, "Adaptable Web Services Modeling using Variability Analysis", in Proceedings of *Third 2008 International Conference on Convergence and Hybrid Information Technology (ICCIT)*, Nov. 11-13, 2008, Busan, South Korea, pp. 700-705.

[11] A. Ruokonen, V. Räisänen, M. Siikarla, K. Koskimies, and T. Systä, "Variation Needs in Service-Based Systems", in Proceedings of *2008 Sixth European Conference on Web Services (ECOWS)*, Nov. 12-14, 2008, Dublin, Ireland, pp. 115-124.

[12] M. Jazayeri, A. Ran, and F.v.d. Linden, *Software Architecture for Product Families: Principles and Practice.* Addison-Wesley, 2000.

[13] M. Svahnberg, J.v. Gurp, and J. Bosch, "A Taxonomy of Variability Realization Techniques", *Software Practice and Experience*, 2005, 35(8): pp. 705-754.

[14] M. Mezini and K. Ostermann, "Variability Management with Feature-Oriented Programming and Aspects", in Proceedings of *the 12th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, Oct. 31-Nov. 6, 2004, Newport Beach, CA, USA, pp. 127-136.

[15] M.J. Escalona and G. Aragón, "NDT. A Model-Driven Approach for Web Requirements", *IEEE Transactions on Software Engineering*, May, 2008: pp. 377-390.

[16] D.C. Schmidt, "Model-Driven Engineering", *IEEE Computer*, Feb., 2006: pp. 25-31.

[17] S. Sendall and W. Kozaczynski, "Model Transformation: The Heart and Soul of Model-Driven Software Development", *IEEE Software*, Sep., 2003: pp. 42-45.

[18] OMG, "Mda Guide Version 1.0.1", Jan. 6, 2003, Available.

[19] G. Karsai, J. Sztipanovits, A. Ledeczi, and T. Bapty, "Model-Integrated Development of Embedded Software", *Proceedings of the IEEE*, 2003, 91(1): pp. 145-164.

[20] J. Greenfield and K. Short, *Software Factories: Assembling Applications with Patterns, Models, Frameworks, and Tools*. Wiley Pub., Indianapolis, IN, USA, 2004.

[21] A. Mattsson, B. Lundell, B. Lings, and B. Fitzgerald, "Linking Model Driven Development and Software Architecture: A Case Study", *IEEE Transactions on Software Engineering*, Oct., 2008: pp. 1-12.

[22] F. Buschmann, R. Meunier, H. Rohnert, P. Sommerlad, and M. Stal, *Pattern-Oriented Software Architecture: A System Of Patterns*. John Wiley & Sons Ltd., West Sussex, England, 1996.

[23] J. Arlow and I. Neustadt, *UML 2 and the Unified Process: Practical Object-Oriented Analysis and Design (2nd Edition) (Addison-Wesley Object Technology Series)*. Addison-Wesley Professional, 2005.