Service Oriented Solution Modeling and Variation Propagation Analysis based on Architectural Building Blocks

Liang-Jie Zhang¹, Jia Zhang²

¹Kingdee International Software Group CO., Ltd, China

²Carnegie Mellon University – Silicon Valley, USA

zhanglj@ieee.org; jia.zhang@sv.cmu.edu

ABSTRACT:

In spite of the widely recognized benefits of applying Service Oriented Architecture (SOA) to design enterprise-scale software systems, its actual application practice is not always a success. 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 new method of architectural building blocks (ABB)-based SOA solution design and it is applicable to any layered or tiered infrastructure. We present the modeling of solution-level architectural artifacts and their relationships, whose formalization enables event-based variation notification and propagation analysis. The goal is to provide architecture-level support for configuration and re-configuration of architectural artifacts based on industry practices. Our method also supports solution-level project variation management for the process of updating and maintaining architectural artifacts. We report a prototype tool that we have developed and describe how we extend the Unified Modeling Language (UML) mechanism to implement the system and enable solution-level enforcement as an example. The prototype has been applied in real projects as an SOA solution modeling tool.

KEY WORDS:

Solution Modeling, Architecture overview diagram, CASE, evolutionary prototyping

INTRODUCTION

Service Oriented Architecture (SOA) has been widely acknowledged as a powerful model that opens a new cost-effective way of engineering software systems, by dynamically integrating and composing existing components into new business processes (Zhang, Zhang et al. 2007). Business applications designed and developed using the SOA technology are called SOA solutions or services solutions.

However, the actual application practice of SOA is not always successful. To date, most of SOA practices are conducted in an *ad hoc* manner and mainly based upon solution architects' personal experiences. This is obviously neither practical nor efficient. Among others, one major reason is the lack of a systematic engineering methodology and an integrated design tool.

Unified Modeling Language (UML) mechanism (Arlow and Neustadt 2005), including Rational Unified Process (RUP) and Rational tools, has been extensively used to help software engineers design and develop enterprise-scale application systems in a uniform way. However, UML tools were designed for building generic software systems. Thus, they do not provide normative guidance and system-level support for building SOA solutions.

From industry practice, layered architectural models have been widely adopted by SOA practitioners to build SOA solutions. A typical example is IBM's Service-Oriented Solution Stack

(S3), which defines a conceptual view of a service-oriented architecture in the solutioning context (Arsanjani, Zhang et al. 2007). As shown in Figure 1, nine layers are identified, organized into a two-dimensional architecture with five horizontal layers and four vertical layers. The horizontal dimension (Operational System layer, Service Component layer, Service layer, Business Process layer, and Solution Consumer layer) implements functional requirements, and the vertical dimension (Integration layer, Data Architecture layer, Quality of Service (QoS) layer, and Governance layer) provides system-support facilities and enablement.

Fig. 1. SOA Solution Stack model.

However, S3 alone is too coarse grained to be advisory for enterprise-level SOA solution design and development. It only provides high-level guidance for developers to design an SOA solution, i.e., in nine layers. To provide a uniform mechanism for building configurable and reusable SOA solutions on top of S3, we introduced a concept of Architectural Building Block (ABB) as the fundamental unit of an SOA solution (Zhang, Zhang et al. 2008). An ABB is a conceptual component that encapsulates internal states and functions and can be configured, extended, and instantiated. Each layer in S3 is comprised of multiple ABBs, which collaborate to carry out all expected activities. Based on industry experience, we recommended a collection of fine-grained ABB templates for each of the nine layers in S3 (Zhang, Zhang et al. 2008, Zhang and Zhang 2009, Zhang and Zhang 2009, Zhang and Zhang 2009).

Considering the adaptive feature of business scenarios, each project may have to configure and customize ABB templates to apt to proprietary requirements and constraints. Some ABBs, however, bear some inter-relationships between them. If an ABB is changed, related ABBs should be changed as well. Take our template for the Solution Consumer layer shown in Figure 2 as an example. A predefined 1:1 relationship exists between the Consumer ABB and the Presentation View ABB. If a project configures a Consumer ABB instance, a Presentation View ABB instance should be created accordingly.

Therefore, our goal is to build a tooling environment supporting SOA solution-based modeling and design. This paper focuses on supporting solution-level design. Our main contributions are three-fold. First, we extend the existing UML tools, by introducing a built-in ABB-based modeling approach, to facilitate in designing SOA solutions. The rationale is that, S3 and ABB templates form a space of architectural artifacts. The modeling environment can guide software architects in building a qualified SOA solution based on predefined schemes and rules. Second, we leverage the power of formal methods (Traore and Aredo 2004) to model solution-level architectural artifacts and their relationships, which formalization enables our tool to conduct rule-based verification of an SOA solution design and conduct variation propagation analysis when changes are made during design time. Note that variation here refers to different configuration and customization from architectural artifact template library during design time. Third, we report our experience and lessons learnt from industry practice. The prototype system is being applied in real projects as an SOA solution modeling tool.

It should be noted that 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. Meanwhile, note the differences between several words used throughout the paper, component, architectural artifact, and architectural building block. A component refers to a comprising part of the structure of an SOA solution with relatively clearly defined structural boundary. For example, a layer in S3 can be viewed as a component. An architectural building block refers to a reusable element in an SOA solution architecture. An architectural artifact is a more generic term referring to either a component or a building block at architectural level.

The remainder of this paper is organized as follows. We will first discuss related work. Using an ABB template for the Solution Consumer layer as an example, we will introduce modeling notations, variation propagation modeling, and solution modeling. Then we will explain our implementation of ABB modeling by extending the UML modeling technique, and will introduce our prototype system and explain how it validates a customized SOA solution. Afterwards, we will depict an overview of our modeling process. Finally, we will conclude the paper and share some future research directions in the area.

RELATED WORK

Our work is a continuous effort based on S3 and S3-based ABB templates. S3 (Arsanjani, Zhang et al. 2007) lacks a way to define elements and details internal of any layer, and it does not provide normative guidance on how to build prototypes of SOA solutions. In addition, since the models stay at very high level and remain coarse grain, solution architects have to design system architecture for each SOA project from scratch based upon personal experiences. This *ad hoc* approach not only wastes valuable human resources, but also usually results in schedule delays.

Our ABB templates (Zhang, Zhang et al. 2008) provide a comprehensive set of architectural building blocks for the Solution Consumer layer in the S3 model based on our practice. 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 project aims to build an engineering tooling environment to guide software architects in constructing an appropriate SOA solution. More importantly, this paper aims to establish a foundation and formalization for the modeling environment to enable solution-level automatic verification of architectural artifacts.

Some researchers have studied variation analysis in Web services development (Jiang, Ruokonen et al. 2005, Kim and Doh 2008, Ruokonen, Räisänen et al. 2008). Variation is well known as variability management in the generic software development process (Jazayeri, Ran et al. 2000). 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 (Svahnberg, Gorp et al. 2005). Gao et al. (Gao, Zhang et al. 2013) propose a fuzzy and utility theory based method for collaborative product design. They adopt an agent-based mechanism to coordinate design-time variation propagation. 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. (Jiang, Ruokonen et al. 2005) highlight the importance of reusing architecture toward developing Web services. They identify a set of variation points from three resources: WSDL

documents, service endpoints, and business logics. Using UML to model two patterns (interface and implementation), they build a prototype tool to help configure variation points in WSDL documents and service endpoints. However, their modeling stays at simple application examples. The relationships between variation points are not examined. In contrast to their work focusing on parameter-level variation points in WSDL documents and service endpoints, our work focuses on building an engineering tool to help manage variations at the SOA solution level.

Kim and Doh (Kim and Doh 2008) 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. In addition, in contrast with their three-tier model staying at a very high level, our modeling is based on industry practice-based S3 model (as shown in Figure 1) that offers a proved guidance for large-scale SOA solution modeling and design.

Ruokonen et al. (Ruokonen, Räisänen et al. 2008) identify a category of variation needs and types relevant for SOA-based systems, based on two dimensions: system concepts and development process. In each of the perspectives of process, subprocess, service, and service interface, variation issues are discussed and a simple eCard service is used to as an example. However, their variation definitions stay at a high level; thus, they do not provide normative guidance for software architects.

Mezini and Ostermann (Mezini and Ostermann 2004) propose a feature-oriented and aspect-oriented modularization approach to manage variability in the context of system families. In contrast to their work focusing on building a language describing low-level features for variability, our work focuses on modeling variations at solution-level architectural artifacts.

Some researchers leverage service usage history to guide SOA solution design. Zhang et al. (Zhang, Ding et al. 2011) use the past query and invocation histories to calculate the user similarity and propose a collaborative filtering algorithm to rank service candidates. Our earlier work (Zhang, Tan et al. 2011) model service usage history in a service social network, and leverage social networking theory and techniques to support context-aware service recommendation. These techniques can be leveraged in our tooling environment to facilitate SOA solution modeling and proactive recommendation.

Model-driven development (MDD) approaches have been widely used in all phases of the lifecycle of software engineering practices to increase software design and development productivity reduce time-to-market in a systematic manner (Sendall and Kozaczynski 2003, Schmidt 2006, Escalona and Aragón 2008). Representative examples include OMG's MDA (OMG 2003), Domain Specific Modeling (DSM) (Karsai, Sztipanovits et al. 2003), and Software Factories (Greenfield and Short 2004) from Microsoft. Different from those component-based modeling approaches focusing on building prototypes from existing models and patterns, our method uses variation analysis of solution architecture design as a driving force to identify architectural artifacts and their relationships. Zhang and Møller-Pedersen (Zhang and Møller-Pedersen 2013) study how to integrate different development tools in a collaborative environment. They propose a method to combine various modeling languages regarding specifying tool integration, and establish tool chains in order to provide tools as a service. OMG's Service Oriented Architecture Modeling Language (SOAML) aims to guide users in describing desired service characteristics (Gebhart, Baumgartner et al. 2010).

Meanwhile, Mattsson et al. (Mattsson, Lundell et al. 2008) have pointed out that traditional MDD cannot automate enforcement of the architecture on detailed design due to its inability to model

architectural design rules. They also emphasize the importance of formally modeling architectural design rules. Our reported work in this paper forms a foundation to model architectural artifacts and relationships, as well as layered verification rules.

Compared with the reported works focusing on modeling and building single Web services, we concentrate on building a formalized model as a foundation for establishing an engineering method for SOA solution design. We aim to enable and facilitate in designing and developing enterprise-scale SOA solutions that may involve many existing service components, business processes, and other artifacts such as rules.

ABB TEMPLATE EXAMPLE

To ease our discussion, Figure 2 shows the recommended ABB configuration for the Solution Consumer layer of S3. Eight ABBs are identified: A *consumer* ABB represents an external user of the Solution Consumer layer. 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 Solution Consumer layer. A *consumer profile* ABB is responsible for getting customer-specific information to be used by the presentation controller for navigation and content presentation purposes. An *access control* ABB provides access to authentication/authorization capabilities to be used by the presentation controller to allow/prevent what content can be presented to the consumer. A *format transformation* ABB is responsible for translation of Query content format required by the Integration layer and to convert content returned from the Integration layer to a consumer response format. A *configuration rule* ABB is responsible for hosting rules that define how the ABBs can be configured based on consumer request scenarios. A *cache* ABB is responsible for temporarily storing consumer interaction-related data to enhance system performance. For detailed ABB definition and configuration please refer to our previous report (Zhang, Zhang et al. 2008).

Fig. 2. Solution Consumer layer ABBs.

FOUNDATIONS FOR ABB MODELING

ABB Modeling

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

paper.

ABB Type and Instance

Definition 1. An *Architectural Building Block (ABB)* is an autonomous component in S3 that encapsulates internal states and functions and can be configured and extended. An *ABB type* refers to every 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.

Each ABB may be instantiated into several objects in a specific SOA solution.

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
$M_m^n \langle A_{ABB}^X, B_{ABB}^Y \rangle$	ABB type A and ABB type B has a m:n mapping relationship
$M_p^q \langle a^{ABB,A}, b^{ABB,B} \rangle$	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)
$P_{M_p^q}^{R_{\mapsto}^t} \langle a^{ABB,A}, b^{ABB,B} \rangle$	protocol between a pair of ABB instances (M_p^q for mapping and $R_{\mapsto}^t$ for propagation)
$P_{M_m^n}^{R_{UML}} \langle A_{ABB}^X, B_{ABB}^Y \rangle$	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

ABB Mapping Relationships

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: (Cheng, Sun et al.), $\{0..*\}$, or $\{1..*\}$.

$$M_m^n \Rightarrow M_m^n \in \{null, M_1^1, M_1^{0..*}, M_1^{1..*}, M_{0..*}^1, \dots\} \quad (R1)$$

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:

$$M_p^q \langle a^{ABB,A}, b^{ABB,B} \rangle \subseteq M_m^n \langle A_{ABB}^X, B_{ABB}^Y \rangle \Rightarrow ((q \subseteq n) \wedge (p \subseteq m)) \vee (M_p^q = null \Rightarrow M_m^n = null) \quad (R2)$$

Definition 3 requires that a cardinality relationship defined between a pair of ABB instances be a subset of the one defined in the corresponding pair of ABB types. For example, if the relationship between a pair of ABB types is defined as $M_1^{0..*}$, the relationship between their ABB instances can only be one of the following possibilities: $(M_1^1, M_1^{1..*}, M_1^{0..*})$.

Variation Propagation Modeling

In this section, we discuss how we model variation propagation requirements between two ABB components.

Propagation Relationship

To maintain loose coupling between ABBs in an SOA solution, we applied the publisher/subscriber design pattern (Buschmann, Meunier et al. 1996) to manage variation propagation and synchronize cooperating ABBs. By notifying all of its subscribers about changes, a publisher enables single-directional change propagation. Each pair of variation publisher/subscriber relationship can be visually represented as a single-directional relationship, by employing a directed arrow pointing from the variation publisher to the variation subscriber. Such a relationship can be denoted by an operator $\mapsto$.

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 implies a logical arrow pointing from one active ABB to the other passive ABB. The active ABB here means that it is independent on changes from the passive ABB; and its changes have to be reflected to or be reacted by the passive ABB. The publisher/subscriber pattern is applied to each such single-directional relationship, by assigning the role of publisher to the active ABB and the role of subscriber to the passive ABB. We thus

define a pair of variation publisher and variation subscriber to each single-directional relationship. When variation happens at the variation publisher, it is propagated to the corresponding variation subscriber through notification. Such a variation publisher/subscriber relationship is enabled by the variation subscriber registering itself at the variation publisher.

An ABB may interact with multiple ABBs; therefore, it may be involved in multiple single-directional relationships. In each such relationship, it may act as either a variation publisher or a variation subscriber. Meanwhile, for a pair of ABBs having bi-directional relationships, each ABB acts as both a variation publisher and a variation subscriber for each other.

As we discussed, ABB types are meant for metamodel and ABB instances are meant for SOA solution modeling. Here we will discuss propagation between ABB types first.

Propagation between ABB Types

Propagation relationships between ABB types are not as important as that between ABB instances. One important reason is that ABB types are for metamodel only; they need to be embodied in actual models and may need to be flexible to be overwritten in accordance with actual conditions. However, properly designed propagation relationships defined between ABB types can be of use 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.

Meanwhile, 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. Since we are only interested in propagation relationships defined for ABB instances, the propagation relationships defined for ABB types are more casual and do not have to be enforced. Thus, we summarize our considerations of propagation relationships between ABB types and ABB instances in Figure 3.

Fig. 3. Propagation relationship specification model.

As shown in Figure 3, metamodels are typically maintained by lead architects and models are constructed by individual SOA architects. Our strategy is that metamodels can utilize 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	Descriptions	Variation management
-----	-------------	--------------	----------------------

	relationships		
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 must be compliant with that defined between the corresponding pair of ABB types. The requirement can be represented as R3:

$$\forall \left(R_{\rightarrow}^t \left\langle a^{ABB,A}, b^{ABB,B} \right\rangle \right) \Rightarrow R_{\rightarrow}^t \left\langle a^{ABB,A}, b^{ABB,B} \right\rangle \subseteq R_{UML} \left\langle A_{ABB}^X, B_{ABB}^Y \right\rangle \quad (R3)$$

The implication of R3 is two-fold. First, the propagation relationship defined between a pair of ABB types should be among those identified in Table 2. Second, the publisher/subscriber relationship between a pair of ABB instances should be assigned using the strategies summarized in Table 2.

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

$$\forall \left(R_{\rightarrow}^t \left\langle a^{ABB,A}, b^{ABB,B} \right\rangle \right) \Rightarrow R_{\rightarrow}^t \left\langle a^{ABB,A}, b^{ABB,B} \right\rangle \subseteq R_{UML} \left\langle A_{ABB}^X, B_{ABB}^Y \right\rangle \quad (R4)$$

SOA Solution Definition

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

Propagation Path

It is common that various kinds of ABB-related variations could happen at runtime. For example, a new ABB instance is created, a method of an existing ABB instance is modified, an ABB instance broadcasts a message, and so on. When a variation occurs, it needs to be propagated to related ABB instances, and in turn be propagated to subsequently corresponding ABB instances. In short, a variation should be propagated to all directly or indirectly related ABB instances. We formally define a propagation path and propagation set as follows.

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 $G_a^P = (PN, PA, a, PSet)$, where PN represents a set of nodes each representing an ABB instance; PA represents a set of directed arcs; a represents the starting point of ABB instance; and $PSet$ represents a set that including all propagation paths starting from a . Each arc links between two nodes if there is a propagation relationship between the two ABB instances. Such an arc starts from the propagation publisher pointing to the propagation subscriber. 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:

$$\begin{aligned} path &= (\vec{N}, \vec{A}, a), \\ \text{where: } \vec{N} &= \{a, N_1, N_2, \dots, N_n\}, \vec{A} = \{R_{\mapsto}^t(a, N_1), R_{\mapsto}^t(N_1, N_2), R_{\mapsto}^t(N_2, N_3), \dots, R_{\mapsto}^t(N_{n-1}, N_n)\} \\ \vec{N} &\subseteq PN, \vec{A} \subseteq PA \end{aligned}$$

Definition 5 implies that:

$$\forall \alpha \in PA \Rightarrow \exists \left(\left(a^{ABB,A} \in N \right) \wedge \left(b^{ABB,B} \in N \right) \wedge \left(R_{\mapsto}^t \left(a^{ABB,A}, b^{ABB,B} \right) \right) \right) \quad (R5)$$

and

$$\begin{aligned} PSet &= \{P_1, P_2, \dots, P_k\}, P_i = \{a, N_1, N_2, \dots, N_n\}, \\ i &\in [1, k], N_j \in PN, j \in [1, n], a \mapsto N_1, N_1 \mapsto N_2, \dots, N_{n-1} \mapsto N_n \end{aligned}$$

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.

Definition 7. A propagation loop in a propagation set indicates an included circular propagation path. Starting from one included ABB instance, a propagation loop will end with the starting ABB instance. It can be denoted by a directed graph $\langle \vec{L}, \vec{LA}, k \rangle$, where:

$$\begin{aligned} \vec{L} &= \{L_1, L_2, \dots, L_l\}, \\ \vec{LA} &= \{R_{\mapsto}^t(L_1, L_2), R_{\mapsto}^t(L_2, L_3), \dots, R_{\mapsto}^t(L_l, L_1)\}, \\ \text{where } \vec{L} &\subseteq PN, \vec{LA} \subseteq PA, k = L_1 \end{aligned}$$

Rationale. We consider a propagation loop in the context of a propagation set instead of a dependency graph. For example, a section of dependency graph, which contains a propagation loop but may never have any variation occurred, should not be part of our consideration.

Generally, there should exist no circles in a propagation set. If existing, a loop should be broken to avoid infinite message passing; or a specific exception handling mechanism should be established to realize manual interruption (usually by a software architect). Let us consider a simple loop. If a change is required for a business goal, it will be propagated to the implantation services, which will in turn be propagated to the realization service components. However, some technology change may be desired at a service component, e.g., change from .NET to Java. Such a change has to be propagated back to the corresponding service and in turn may cause some change at the business goal. Such a loop is needed; and after a decision is made by coordination agents, the propagation circle will end.

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:

$P_{M_m^n}^{R_{UML}}(A_{ABB}^X, B_{ABB}^Y)$, 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 $M_m^n(A_{ABB}^X, B_{ABB}^Y)$; 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:

$P_{M_p^q}^{R_{\mapsto}^t}(a^{ABB,A}, b^{ABB,B})$, 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 $M_p^q(a^{ABB,A}, b^{ABB,B})$; 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.

$$P_{M_m^n}^{R_{UML}}(A_{ABB}^X, B_{ABB}^Y) \prec P_{M_p^q}^{R_{\mapsto}^t}(a^{ABB,A}, b^{ABB,B}) \Rightarrow (R_{\mapsto}^t \subseteq R_{UML}) \wedge (M_p^q \subseteq M_m^n) \quad (R6)$$

Definition of an SOA Solution

With the preparation of these notations, we are ready to define the notation 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:

$$\forall p \in \Phi_P \Rightarrow$$

$$(\exists A \in \Phi_\Phi) \wedge (\exists B \in \Phi_\Phi) \wedge \left(P_{M_m^n}^{R_{UML}}(A_{ABB}^X, B_{ABB}^Y) \right)$$

- 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:

$$\forall p \in \Omega_P \Rightarrow \left(\exists a^{ABB,X} \in \Omega_\Omega \right) \wedge \left(\exists b^{ABB,Y} \in \Omega_\Omega \right) \wedge \left(P_{M_m^n}^{R_t} \left(a^{ABB,X}, b^{ABB,Y} \right) \right)$$

IMPLEMENTING 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 (Arlow and Neustadt 2005). 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.

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 sets 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 thus will not appear between ABB instances (e.g., instantiation, binding, package import, element import, and association class). 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 (directed association, aggregation, composition, generalization, implements, realization, dependency, and usage) 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. In more detail, an *association* relationship will be broken into two *directed association* relationships; a *link* will also be broken into two. Because from the component-level relationship perspective, the two relationships *association* and *link* act as the same, without losing generality, one *link* relationship will be broken into two *directed association* relationships.

Meanwhile, from the consideration of component-level variation propagation, some relationships will cause the same or similar action. For example, an *aggregation* and a *composition* relationship both represent a whole-part relationship, and a change in a part (e.g., adding a new part or removing an existing part) needs to be propagated to the whole component. For another example,

an *implements* and a *realization* relationship both imply that one component represents an abstract of the other one. Thus, we organize the studied relationships into four categories as shown in Table 2. The detailed definition for each discussed UML relationship can be found from UML 2.0 specification (Arlow and Neustadt 2005) 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*.

As shown in Table 2, 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 (Zhang, Zhang et al. 2007). 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 and importance of differentiating variation levels. To address this need, our solution is to extend the original publisher/subscriber pattern by associating variation level information. In more detail, 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.

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 yet another 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.

Thus, an SOA solution can be modeled using extended UML. Figure 4 shows a segment of a class diagram of the Operational System layer of a particular travel booking system. As shown in Figure 4, two components “Car Reservation Application” and “Hotel Reservation Application” each represent an interface deriving from a *LegacyApplication* ABB stereotype; a component “JVM” (Java Virtual Machine) represents an interface deriving from a *RuntimeEnvironment* ABB stereotype, a component “Operational System Controller” represents an instance deriving from a *OperationalSystemController* ABB stereotype.

Fig. 4. A segment of class diagram.

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 as what we originally did, 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.

Algorithm 1: Metamodel-centered SOA solution model management

1. If not existed, 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 stereotypes (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 stereotypes (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 the corresponding model library elements only.

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.

Example of UML Extensions for Solution Consumer layer

We will use the Solution Consumer layer as an example to illustrate how we use metamodel to simplify and automate stereotype management, by using metamodels as the base to enable automatic generation of profiles and palettes. As a prerequisite for our method, we create a S3 base profile serving all nine layers in the S3 model. The central stereotype in the S3 base profile is the stereotype *ABB*. Based on the S3 base profile, as shown in Figure 5, we create a metamodel to define the eight types of ABBs and their relationships for the Services Consumer layer as defined in Figure 3. As shown in Figure 5, the metamodel is created as a class diagram containing a set of eight classes, each representing an identified ABB. Each class is marked as an instance of the stereotype *ABB*.

A profile will be automatically generated for the Solution Consumer layer. Each ABB depicted in Figure 5 becomes a UML stereotype. The profile of the Solution Consumer layer thus comprises eight stereotypes. Each stereotype derives from basic UML 2.0 stereotype *class*. Each layer in S3 model possesses a proprietary profile. Note that such a stereotype generated will only allow a creation of its instance extending from UML stereotype *class*. If we expect to create an instance of the generated stereotype extending other UML stereotypes, such as *interface*, we will have to manually add a corresponding super stereotype in the profile. This implication comes from the fact that a metamodel is typically created as a UML class diagram comprising only classes, which is the typical way to represent data types (i.e., stereotypes). Thus, this limitation is in general tolerable.

Fig. 5. Services Consumer layer metamodel.

As shown in Figure 6 on the left panel, a model library will be automatically created for the Services Consumer layer. Each ABB depicted in Figure 5 becomes a building block in Figure 6. Note that at this moment, the profiles created before have been applied. Each of the building blocks in the Services Consumer layer has already been marked as representing the corresponding stereotype. For the building block of “consumer” highlighted in Figure 6 on the left panel, the stereotype “Consumer” has already been applied. Meanwhile, each generated building block is marked with the signal “\$” and is delimited with a pair of brackets, which implies that when a user creates an instance of the building block, he/she should change the name of the instance to

Fig. 6. Generated default modeling template

specific name. Moreover, Figure 6 shows that the each layer in S3 model possesses a proprietary model library.

Usages of Metamodel for Generating Solution Patterns

The industry practice we use not only leads to the identification of ABBs, but also depicts the relationships between the ABBs. As shown in Figure 5, 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 6 shows an example structure of a Services Consumer layer of an SOA solution. As shown in Figure 6, each of the ABB identified has an instance; these ABB instances are interconnected according to the relationships defined in Figure 5.

Meanwhile, based on industry practice, our metamodels also implicates a design pattern of the corresponding layers in the S3 model. For example, as shown in Figure 6, by instantiating every stereotype in the metamodel in Figure 5 and mapping the relationships between stereotypes to their instances, we can obtain a modeling template of the Services Consumer layer. Similar to that a design modeling template representing a reusable solution to a repeatable scenario, this pattern represents a starting point to build a customized Services Consumer layer for a specific SOA solution.

Therefore, we propose another novel way to utilize our metamodels, by automatically generating a modeling template for each layer in the S3 model. The main concept of the algorithm is similar to the one introduced in earlier section. The difference is that, this time in addition to iterating through each identified ABB in a metamodel, we will also catch all relationships. For every ABB caught, a class instance will be created as an instance of the stereotype; for every relationship

caught between two stereotypes, a relationship will be created between the corresponding two class instances.

SOA SOLUTION VALIDATION

After we introduce the basic modeling aspects of an SOA solution, in this section, we discuss how to validate a customized SOA model, through layered architectural rule sets.

Hierarchical Rule Sets

We organize architectural rules in a three-level hierarchy: (1) top-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. For example, the Solution Consumer layer may allow only one instance of *Presentation Controller* ABB. ABB-level rules refer to the relationships defined between ABB types that have to be enforced on corresponding ABB instances. For example, assume a one-to-many dependency relationship is defined between ABB type *A* and *B*. If an instance of ABB type *B* is created, it should be registered in the instance of ABB type *A*.

System-level Rules

The global rule set contains six default rules as defined in Definition 12.

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

1. $\forall A_{ABB}^X \Rightarrow \exists (\Phi_X \in \Phi_\Phi) \wedge (A_{ABB}^X \in \Phi_X)$
 $\wedge (\exists A_{ABB}^X \wedge \exists A_{ABB}^Y \Rightarrow X = Y)$
 $\forall a^{ABB,A} \Rightarrow \exists (\Omega_X \in \Omega_\Omega) \wedge (a^{ABB,A} \in \Omega_X) \wedge$
2. $(\exists a^{ABB,A} \in \Omega_X \wedge a^{ABB,A} \in \Omega_Y \Rightarrow X = Y)$
3. $\Phi_X = null \Rightarrow \Phi_X = \phi$
4. $\Phi_X = \phi \Rightarrow \Omega_X = \phi$
5. $*_A^X \geq 0$
6. $\forall \left(P_{M_m^n}^{R_{UML}} \langle A_{ABB}^X, B_{ABB}^Y \rangle \wedge P_{M_p^q}^{R_t} \langle a^{ABB,A}, b^{ABB,B} \rangle \right) \Rightarrow$
 $P_{M_m^n}^{R_{UML}} (A_{ABB}^X, B_{ABB}^Y) \prec P_{M_p^q}^{R_t} (a^{ABB,A}, b^{ABB,B})$

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. Specially, rule 6 also implies that: (1) if there is no cardinality relationship between a pair of ABB types, there should be no cardinality relationship between the corresponding pair of ABB instances; (2) if there is cardinality relation between a pair of ABB types, their corresponding ABB instances have to be connected with the same or stronger cardinality relationships; and (3) if there is propagation relation between a pair of ABB types, their corresponding ABB instances have to use the way summarized in Table 2 for assigning publisher and subscriber roles.

In general, the top-level rules are implicitly defined and cannot be modified throughout the modeling of any SOA solution.

Layer-level Rules

Each layer in the S3 model can carry a specific set of rules as layer-level rules. Taking the Solution Consumer layer as an example, its layer-level rule set possesses one rule R7:

$$\Omega_{Consumer} \neq \phi \Rightarrow \text{iff} \left(\begin{array}{l} presentation_controller^{ABB, Presentation_Controller} \\ \in \Omega_{Consumer_Layer} \wedge *_{presentation_controller}^{Consumer_Layer} = 1 \end{array} \right) \quad (R7)$$

R7 denotes that if the Solution 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 Solution Consumer layer. This rule can be represented as R8 and can be set up by setting the multiplicity of the *Presentation Controller* ABB type to be “1..*”.

$$\Omega_{Consumer} \neq \phi \Rightarrow \text{iff} \left(\begin{array}{l} presentation_controller^{ABB, Presentation_Controller} \\ \in \Omega_{Consumer_Layer} \wedge *_{presentation_controller}^{Consumer_Layer} \geq 1 \end{array} \right) \quad (R8)$$

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:

$$\Omega_{Presentation_Layer} \wedge *_{Consumer}^{Presentation_Layer} \geq 1 \quad (R9)$$

ABB-level Rules

ABB-level rules define the mutual connections between ABB types that need to be enforced on corresponding ABB instances. As defined in earlier section, 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 one-to-zero_to_many dependency relationship between a *Presentation Controller* ABB type and an *Access Controller* ABB type in the Solution Consumer layer, which can be represented as follows:

$$P_{M_1^{0..*}}^{R_{Dependency}} \left(\begin{array}{c} \text{Presentation_Controller}_{ABB}^{Consumer_Layer} \\ \text{Access_Controller}_{ABB}^{Consumer_Layer} \end{array} \right), \quad (R10)$$

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

$$\begin{aligned} (1) & P_{M_1^{0..*}}^{R_{\rightarrow}} \left(\begin{array}{c} \text{presentation_controller}_{ABB, Consumer_Layer} \\ \text{access_controller}_{ABB, Consumer_Layer} \end{array} \right) \\ (2) & P_{M_1^{1..*}}^{R_{\rightarrow}} \left(\begin{array}{c} \text{presentation_controller}_{ABB, Consumer_Layer} \\ \text{access_controller}_{ABB, Consumer_Layer} \end{array} \right) \\ (3) & P_{M_1^1}^{R_{\rightarrow}} \left(\begin{array}{c} \text{presentation_controller}_{ABB, Consumer_Layer} \\ \text{access_controller}_{ABB, Consumer_Layer} \end{array} \right) \end{aligned}$$

The deduced 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 deduced 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.

Note that the publisher/subscriber relationship is an implementation-oriented strategy considering variation adaptations. An SOA architect also uses common semantic-rich relationships (e.g., UML 2.0-defined relationship types) to define the relationships between ABB instances when creating an SOA solution. To enable validation and verification of a user-created model, our strategy is to translate the relationship between each pair of ABB instances into a publisher/subscriber relationship, using the algorithms defined in Table 2 for each type of UML 2.0 relationship. For example, if there is a one-to-zero_to_many dependency relationship defined between a *Presentation Controller* ABB instance and an *Access Controller* ABB instance in the Solution Consumer layer, it can be transformed into the following relationship:

$$P_{M_1^{0..*}}^{R_{\rightarrow}} \left(\begin{array}{c} \text{presentation_controller}_{ABB, Consumer_Layer} \\ \text{access_controller}_{ABB, Consumer_Layer} \end{array} \right)$$

Comparing to the deduced ABB instance rules knowledge base above, it can be seen that the defined relationship can be accepted. For another example, if there is a one-to-zero_to_many dependency relationship defined between an *Access Controller* ABB instance and a *Presentation Controller* ABB instance in the Solution Consumer layer, it can be transformed into the following relationship:

$$P_{M_1^{0,*}}^{R_{\rightarrow}} \left(\begin{array}{l} access_controller^{ABB, Consumer_Layer}, \\ presentation_controller^{ABB, Consumer_Layer} \end{array} \right)$$

Obviously the created relationship cannot be found from the deduced rules; therefore, it should be rejected.

In summary, each relationship defined between a pair of ABB types in a metamodel can be deduced 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 deduced rule sets. If it can be found in the knowledge base, it is a valid relationship; otherwise, it should be rejected.

MODELING ENVIRONMENT AND PROCESS

With the establishment of ABB-based modeling technique and solution validation mechanism, we have constructed SOA Modeling Environment (SOA-ME) as a prototype modeling tool. The framework of the tool is shown in Figure 7. It is built upon IBM Rational Software Architect (RSA) (IBM) 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 contains three components (metamodel parser, layer-level rule parser, and rule generator) and two databases (system-level rules and rule knowledge base). The *metamodel parser* takes input of a metamodel and conducts initial analysis over the model. The *layer-level rule parser* takes input of layer-level rules and conducts initial analysis. Both the metamodel parser and the layer-level rule parser output results to the rule generator, which also takes input from the predefined system-level rules database. The rule generator thus takes input from the three sources and creates system-level rules, layer-level rules, and ABB-level rules into the *rule knowledge base*.

The static validation section contains two components (static model parser and propagation analyzer) and two databases (model rules and propagation sets). The *static model parser* takes input of a customized SOA model, analyzes its contents, while checking the rule knowledge base. In the process of the static model validation, as discussed in the previous sections, the *propagation analyzer* generates normalized propagation sets and stores them into the *propagation sets* database. The result of the static model parser is valid SOA model, which can be deployed for runtime usages.

The dynamic validation section contains one component: dynamic model parser. When variations happen at runtime, the *dynamic model parser* examines the messages passed between ABB instances and re-engineers the propagation paths. Then it compares the generated sets with the

Fig. 7. S3-Modeling Environment (SOA-ME).

stored propagation paths created at the static validation phase. If the two sets are equivalent, the validation result is positive.

Conclusions

In this paper, we present our design and development of a systematic SOA solution modeling and variation propagation analysis based on the industry practice-based infrastructure 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. The formalization of variation-oriented analysis for architectural artifacts allows solution-level enforcement and performance analysis, and supports architectural evolutionary changes.

Although our ABB-based modeling is currently implemented using the UML technique, it is not limited to the method. However, considering its powerful modeling and tooling support, UML is currently one of the recommended realization techniques. We also use UML to validate our formalized approach.

Currently, our prototype modeling innovation is part of the IBM SOMA Modeling Environment

(SOMA-ME) (Zhang, Zhou et al. 2008), which is being used by a variety of industry projects as the core tool to conduct SOA solutioning services since 2006 to win big bids and effectively deliver SOA solution design services.

For our future study, we plan to study configurable 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. It is noted that the presented modeling approach can be applied to any layered or tiered architecture.

REFERENCES

- Arlow, J. and I. Neustadt (2005). UML 2 and the Unified Process: Practical Object-Oriented Analysis and Design (2nd Edition) (Addison-Wesley Object Technology Series), Addison-Wesley Professional.
- Arsanjani, A., L.-J. Zhang, M. Ellis, A. Allam and K. Channabasavaiah (2007). "S3: A Service-Oriented Reference Architecture." IT Professional: 10-17.
- Buschmann, F., R. Meunier, H. Rohnert, P. Sommerlad and M. Stal (1996). Pattern-Oriented Software Architecture: A System Of Patterns, John Wiley & Sons Ltd., West Sussex, England.
- Cheng, H.-T., F.-T. Sun, S. Buthpitiya and M. L. G. 2010: (2010). SensOrchestra: Collaborative Sensing for Symbolic Location Recognition. International Conference on Mobile Computing, Applications, and Services (MobiCASE), Santa Clara, CA, USA.
- Escalona, M. J. and G. Aragón (2008). "NDT. A Model-Driven Approach for Web Requirements." IEEE Transactions on Software Engineering: 377-390.
- Gao, S., S. Zhang, X. Chen and Y. Yang (2013). "A Framework for Collaborative Top-Down Assembly Design." Computers in Industry **64**(8): 967-983.
- Gebhart, M., M. Baumgartner, S. Oehlert, M. Bliersch and S. Abeck (2010). Evaluation of Service Designs based on SOAML. 5th International Conference on Software Engineering Advances (ICSEA), Washington DC, USA.
- Greenfield, J. and K. Short (2004). Software Factories : Assembling Applications with Patterns, Models, Frameworks, and Tools. Indianapolis, IN, USA, Wiley Pub.
- IBM. "Rational Software Architect (RSA)." from <http://www-01.ibm.com/software/awdtools/architect/swarchitect/>.
- Jazayeri, M., A. Ran and F. v. d. Linden (2000). Software Architecture for Product Families: Principles and Practice, Addison-Wesley.
- Jiang, J., A. Ruokonen and T. Systa (2005). Pattern-Based Variability Management in Web Service Development. Third IEEE European Conference on Web Services (ECOWS), Växjö, Sweden.
- Karsai, G., J. Sztipanovits, A. Ledeczi and T. Bapty (2003). "Model-Integrated Development of Embedded Software." Proceedings of the IEEE **91**(1): 145-164.
- Kim, Y. and K.-G. Doh (2008). Adaptable Web Services Modeling using Variability Analysis. Third 2008 International Conference on Convergence and Hybrid Information Technology (ICCIT), Busan, South Korea.
- Mattsson, A., B. Lundell, B. Lings and B. Fitzgerald (2008). "Linking Model Driven Development and Software Architecture: A Case Study." IEEE Transactions on Software Engineering: 1-12.
- Mezini, M. and K. Ostermann (2004). Variability Management with Feature-Oriented Programming and Aspects. the 12th ACM SIGSOFT International Symposium on Foundations of Software Engineering, Newport Beach, CA, USA.

OMG. (2003, Jan. 6). "Mda Guide Version 1.0.1."

Ruokonen, A., V. Räisänen, M. Siikarla, K. Koskimies and T. Systä (2008). Variation Needs in Service-Based Systems. 2008 Sixth European Conference on Web Services (ECOWS), Dublin, Ireland.

Schmidt, D. C. (2006). "Model-Driven Engineering." IEEE Computer: 25-31.

Sendall, S. and W. Kozaczynski (2003). "Model Transformation: The Heart and Soul of Model-Driven Software Development." IEEE Software: 42-45.

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

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

Zhang, J., W. Tan, J. Alexander, I. Foster and R. Madduri (2011). Recommend-As-You-Go: A Novel Approach Supporting Services-Oriented Scientific Workflow Reuse. IEEE International Conference on Services Computing (SCC), Washington DC, USA.

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

Zhang, L.-J. and J. Zhang (2009). Design of Service Component Layer in SOA Reference Architecture. Proceedings of 33rd Annual IEEE International Computer Software and Applications Conference (COMPSAC 2009), Seattle, WA, USA.

Zhang, L.-J. and J. Zhang (2009). "An Integrated Service Model Approach for Enabling SOA." IEEE IT Professional **11**(5): 28-33.

Zhang, L.-J., J. Zhang and A. Allam (2008). A Method and Case Study of Designing Presentation Module in An SOA-based Solution Using Configurable Architectural Building Blocks (ABBs). IEEE International Conference on Services Computing (SCC), Honolulu, HI, USA.

Zhang, L.-J., J. Zhang and H. Cai (2007). Services Computing, Springer.

Zhang, L.-J., N. Zhou, Y.-M. Chee, A. Jalaldeen, K. Ponnalagu, R. R. Sindhgatta, A. Arsanjani and F. Bernardini (2008). "SOMA-ME: A Platform for the Model-Driven Design of SOA Solutions." IBM Systems Journal **47**(3): 397-413.

Zhang, Q., C. Ding and C. H. Chi (2011). Collaborative Filtering based Service Ranking using Invocation Histories. IEEE International Conference on Web Services (ICWS), Washington DC, USA.

Zhang, W. and B. Møller-Pedersen (2013). Establishing Tool Chains above the Service Cloud with Integration Models. IEEE International Conference on Web Services (ICWS), Santa Clara, CA, USA.

ABOUT THE AUTHORS

Liang-Jie (LJ) Zhang, Ph.D., is Senior Vice President, Chief Scientist, & Director of Research at Kingdee International Software Group Company Limited, and director of The Open Group. Prior to joining Kingdee, he was a Research Staff Member at IBM Thomas J. Watson Research Center. Dr. Zhang has published more than 140 technical papers in journals, book chapters, and conference proceedings. He has 40 granted patents and more than 20 pending patent applications. Dr. Zhang received his Ph.D. on Pattern Recognition and Intelligent Control from Tsinghua University in 1996. He chaired the IEEE Computer Society's Technical Committee on Services Computing from 2003 to 2011. He also chaired the Services Computing Professional Interest Community at IBM Research from 2004 to 2006. Dr. Zhang has served as the Editor-in-Chief of the International Journal of Web Services Research since 2003 and is the founding Editor-in-

Chief of IEEE Transactions on Services Computing. He was elected as an IEEE Fellow in 2011, and in the same year won the Technical Achievement Award "for pioneering contributions to Application Design Techniques in Services Computing" from IEEE Computer Society. Dr. Zhang also chaired the 2012 IEEE 1st International Conference on Services Economics (SE 2012) and the 2009 IEEE International Conference on Cloud Computing (CLOUD 2009). He can be reached at zhanglj@ieee.org.

Jia Zhang, Ph.D., is an Associate Professor at Carnegie Mellon University – Silicon Valley. Her current research interests center on Services Computing, with a focus on collaborative scientific workflow, service-oriented architecture, semantic service discovery, and Internet of Things. She has co-authored one textbook titled “Services Computing” and published over 120 journal articles, book chapters, and conference papers. Zhang is an associate editor of IEEE Transactions on Services Computing (TSC) and of International Journal on Web Services Research (JWSR). She is a member of the IEEE and can be reached at jia.zhang@sv.cmu.edu.