2009 IEEE International Conference on Web Services

Contract-First Design Techniques for Building Enterprise Web Services

Youliang Zhong
Freelance consultant
Epping, NSW, Australia
ylzhong@optusnet.com.au

Jian Yang
Macquarie University
North Ryde, NSW, Australia
jian@ics.mq.edu.au

***Abstract*—Based on real development experience, the paper presents a collection of design techniques for building enterprise web services. By applying the techniques to web services development, not only the development increases reusability and productivity, but also the web services improve agility and compatibility.**

Enterprise web services require high grade of competency in designing web service contracts. A contract of web service formalizes an agreement between web service provider and consumer, in the forms of WSDLs, service schemas and policies. Though contract-first method provides great potential of directly dealing with the contracts, and a number of articles have been published regarding designing WS and XML schemas, however it is still hard for developers to find cookbooks or guidelines concentrated on designing web service contracts with contract-first method. To fill the gap, a set of design techniques are introduced and deployed in practice, incorporating some best practices scattered over the web services community. These techniques cover most of the key aspects of web service, including consolidating service schemas in line with business entities, constructing coarse-grained namespaces, applying versioning over WSDLs and service schemas, and writing fine-grained filters with contracts.

I. Introduction

With its simplicity and interoperatibility, nowadays web services technology has become one of the focal points of enterprise software development. It was reported that, when questioned about ”select the most important trend impacting your business” regarding enterprise software trends in 2008, the answers gave a rate of 25% to Web Services and SOA [1]. That shows web services have received great appreciation from the majority of enterprises.

Enterprise web services represent enterprise systems that are built by web services technology. Generally speaking, building enterprise web services needs to handle numerous business entities, stable and compatible evolution, and fine-grained policies. Developers need to pay more efforts to design web service contracts to deliver high quality enterprise web services.

A contract of web service is an agreement between service provider and consumer, which formalizes the details of the service (contents, protocols, delivery process, quality criteria, etc) in a way that meets mutual understandings and expectations of both parties. From a technical perspective, a contract of web services is described in the forms of WSDLs, service schemas and policies.

Code-first and contract-first are two basic methods of building web services. Code-first methods develop programming code first, generate WSDLs with service schemas from the code, and then publish the generated WSDLs and the programming code to make services available [2], [3]. Code-first approach is especially adequate for bottom-up delivery strategy when building web services on an as-needed basis.

When build enterprise web services by code-first methods, there is nearly no room for developers to design web service contracts. The following issues against code-first methods have been addressed [4], [6].

- Object/XML mismatching: Mismatches between Object and XML often occur when generate WSDLs and service schemas from programming code, which hurt the expected interoperatibility of web services across multiple platforms.
- Less reusability of service schemas: Because WSDLs and service schemas are generated from individual methods or classes each time, therefore same schema elements are often duplicated within different WSDLs or schema files.
- Too-fine-grained namespaces: Because of the above reason, whenever a WSDL and related service schemas are generated, some namespaces are mechanically created. As a result, too-many and too-fine-grained namespaces are produced, which make change management and versioning difficult.
- Poor compatibility of services: Owing to the causes stated above, two more issues come out. Firstly the compatibility of services is easily damaged owing to incompatible programming code. Secondly it is difficult to have generated WSDLs and service schemas keep compatibility information between different versions.

Opposite to code-first methods, various contract-first methods came out in recent years [6]–[11]. As the name indicates, contract-first methods develop web service con-

978-0-7695-3709-2/09 $25.00 © 2009 IEEE
DOI 10.1109/ICWS.2009.91

tracts first before write programming code. By doing so, developers get great power of designing web service contracts, potentially in the following service layers [12].

- service structural layer
  The contents in service structural layer are operations, messages, endpoints, information types, etc. The design work at this layer includes modeling data and messages, organizing service schemas, designing namespaces and versions of service schemas, and etc.
- service behavioral layer
  This layer consists of two aspects, one is service composition such as operation sequence and state, the other service connectivity such as service addressing and binding. The tasks in the layer include binding services and port types, configuring namespaces and versions of services, and so on.
- service regulatory layer
  The service regulatory layer defines business rules and various policies of services. At this layer the focus of design work is shifted from designing artifacts to wiring them up.

Contract-first methods have overcome some of the above issues, also there are various industry efforts and articles published over the conferences and internet about designing WSDL and Schema. However it is hard to find cookbooks or guidelines which dedicate to well-established design techniques to build contract-first web services.

Incorporating some of best practices over the web services community, this paper presents a number of methodologies and guidelines and shows how to put them into practice. These techniques help developers design the contracts of enterprise web services, which are highlighted as follows.

- Design of Service Schema: This is essentially the design work at service structural layer, including the following tasks: separating service schemas from WSDLs, removing duplicated schema elements, organizing service schemas in line with business entities, and making service schemas stand between WSDLs and business entities. By doing so, the service schemas get great reusability, meanwhile reduce duplication and inconsistency.
- Design of Namespace: Namespace is one of the most important aspects of service connectivity in both structural and behavioral layers. The design work of namespace includes such techniques as grouping related functions, introducing global namespaces and using proxy schema. As a result, coarse-grained namespaces not only reduce the burden of change management, but also achieve the principle of Separation of Concerns or SoC.
- Design of Service Versioning: Service versioning affects both layers of service structure and service behavior. Well-structured versioning needs guidelines of how to configure multiple minor version services within WSDL. The versioning techniques improve service compatibility and help service evolution.
- Design of Filter: Filters are implementation of policies defined in service regulatory layer. The design work of filters has two tasks. One is to write filters by collaborating service schemas, and the other to make filters fine-grained at business entity level. That can be realized by using AOP technique [20].

The followings sections elaborate the details of the listed techniques. Section II outlines the background of a motivating case study, and section III exemplifies the details of the techniques. Section IV discuss related work. The paper is concluded in section V.

II. Motivating Case Study

The case study under discussion [1] is an enterprise system, originally built by PowderBuilder technology with Sybase database for more than one decade. The system has been successfully deployed in a number of client sites. Years ago, a project was setup to build a web GUI of administration functionality, as well as an API to be accessed from various platforms.

The project decided to use Java-based web services technology on the top of existing database. The first goal was to deliver 40 more functions as web services. For example, relating to a business entity *Account*, five functions were to become web services: *Set-Account*, *Get-Account-Summary*, *Get-Account-Details*, *Update-Account* and *Delete-Account*.

The development followed the contract-first method of Spring-WS. The team developed WSDLs and service schemas first, then used tools to generate data transfer objects or DTOs [2] from service schemas, and the DTOs were further mapped to business entities in database. The development successfully designed WSDLs, service schemas and filters by incorporating a number of design techniques. Particularly three efforts were paid to designing web service contracts.

The first and most effort was made to service schema design. At initial stage, because each function corresponded to one WSDL, and each WSDL had a pair of input and output messages, thus produced two service schema files or schemas. As a result, totally 80 more schemas were produced. Of them, there were a lot of duplications. Also there were multiple elements defined for a single business entity. Followed step-by-step design techniques, the service schemas were finally consolidated by 8 schemas, grouped by business domains such as Accounting, Product, Ordering, and so on.

Secondly great attention was paid to constructing coarse-grained namespaces. As stated above, each WSDL had a pair

[1]The following functions and code snippets of the case study are especially prepared for exemplification purpose used in the paper.

[2]Data Transfer Object or DTO is such an object that does not have any behavior except for storage and retrieval of its own data [21].

of schemas, which were assigned different namespace, thus the number of namespaces reached more than 80. Besides the burden of change management, fine-grained namespaces made versioning too complicate. To solve the issue, two techniques were adopted. One is grouping related functions, and the other using proxy schema. By doing so, the total number of namespaces was reduced to 20.

The project applied major-minor versioning to WSDL and service schemas, which made it possible to compose various services in an agile and flexible way, so that increased client satisfaction. Furthermore, carefully design minor-versions brought least impact to service evolution.

The success of the case study proved that the design techniques in the paper are effective and pragmatic, which we will share with readers in the following sections.

III. Design Techniques

This section exemplifies the design techniques in four aspects: service schemas, namespaces, versioning and filters.

A. Consolidate service schemas

Under the circumstances of enterprise web services, service schemas become the major part of service infrastructure. The concern is over the mapping between service schemas and business entities. Usually it takes four steps as follows.

1) Remove duplicated schemas: In most textbooks, service schemas are often embedded in <types> section of a WSDL. However WSDL specification [13] also provides an 'import' mechanism that makes it possible to separate service schemas from WSDLs.

There are always a lot of schema elements duplicated in different schema files. Taking an example in the case study, two functions *Get-Account-Summary* and *Get-Account-Details* associated two messages which pointed to two schema elements: *GetSummary* and *GetDetails*. These elements were further defined by two schema elements: *GetAccountSummary* and *GetAccountDetails*. Looking into them, both point to a same element: *AccountId*. So removing such duplicated schemas becomes a task of consolidating service schemas.

2) Common schemas: A common practice in developing service schemas is that individual developers may create multiple schema elements for a same business entity, but by different definitions. For instance, in the case study, there were several schema elements such as *Sex*, *Gender*, *SexStatus* and etc for a single business entity *GENDER*, each had different definition.

To make greater reuse, also to remove inconsistency, a common schema can be created say Base-Data-Types that includes most commonly-use data types. For those modifiers or extensions that are specific to business logic, there needs another parallel common schema say Domain-Data-Types.

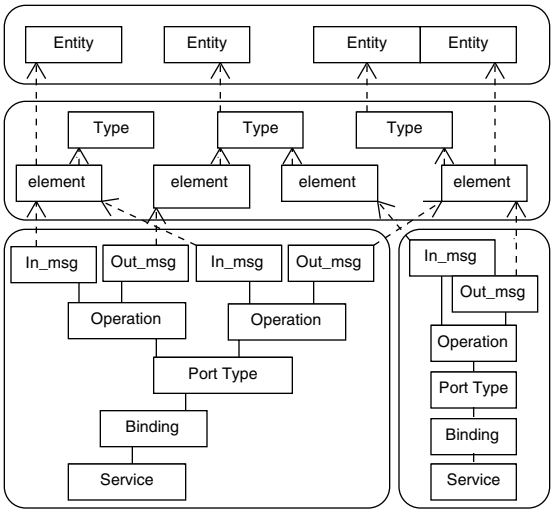

Figure 1. Line with business entities

For instance, there is a schema element *Percentage* in Base-Data-Types, however the business logics of Investment and Payment need extension on the top of *Percentage*. Therefore two schema elements *InvestmentPercentage* and *Payment-Percentage* are defined in Domain-Data-Types. A discipline of schema design is to reuse these common schemas as far as possible.

3) Line with business entities: Though there seems no unique answer about how to organize service schemas, the approach in the case study would be a good reference. That is, besides the common schemas stated above, all other schema elements are divided into two groups: *Request* and *Response*. In *Request* group, elements are organized by three schemas: *Enquire*, *Update* and *Deletion*. The schema *Update* can include *Set* functions, because in most cases the contents of *Update* and *Set* are the same or similar. In *Response* group, elements are classified according to business domains such as *Accounting*, *Ordering*, *Customers*, *Products* and so on.

Experiences show that dividing service schemas into *Request* and *Response* helps maintaining web service backward compatibility [18]. That also helps constructing coarse-grained namespaces.

4) Stand between WSDLs and business entities: By applying the above techniques, service schemas become a global base for all WSDLs. Such service schemas become a middle layer between WSDLs and business entities, as showed in Figure 1. Thereby service operations are merely various behavior aspects of business entities. That coincides with one of the goals of web services: web services can be dynamically delivered to meet business agility, based on relatively stable business entities.

In the case study, developers prepared service schemas by consulting functional specification and database definitions,

while business analysts reviewed the schemas to make sure if they satisfied the requirements or not. That is exactly the ideal meet-in-the-middle delivery strategy [5] where business and system people work together to deliver high quality web services.

The above steps form up a well-structured approach for service schema development, which is specially helpful for those projects which start from scratch or need fundamental changes in schemas. An alternative way is creating schemas-entities mappings to achieve the same goals. Regardless which way to go, the above Base-Data-Types and Domain-Data-Types are most important. If these schemas are well constructed, the further development will become quicker and cheaper than ever.

B. Construct coarse-grained namespaces

Namespace defines a scope of a set of element and attribute names [14], so that serve the modularity of web services. The design goal here is to construct coarse-grained namespaces for WSDLs and service schemas. To achieve this goal, there are three techniques or tasks.

1) Group related functions: Most code-first methods ask users to assign an 'Interface method', and then generate a WSDL and service from the method. That results in a design pattern of one-function-one-WSDL.

Taking an example of the five functions listed in previous section II: *Set-Account*, *Get-Account-Summary*, *Get-Account-Details*, *Update-Account* and *Delete-Account*. When they are designed by one-function-one-WSDL pattern, five different packages are created, thus duplicated classes are inevitably produced in the five packages.

To avoid this, one has to group related functions into a single WSDL if they are related to same business entities. Doing so, the number of WSDL namespaces will be significantly reduced. Coarse-grained namespaces not only reduce the burden of change management, but also make versioning easier.

2) Introduce global namespaces: There is no standard answer regarding how to assign namespaces to service schemas. According to the experience in the case study, the service schemas are firstly grouped in three categories: *Common*, *Request* and *Response*. The *Common* group consists of Base-data-types and Domain-data-types. The *Request* group includes *Enquire*, *Update* (including *Set* functions) and *Deletion* schemas. At last, the schemas in *Response* group are classified by business domains such as *Accounting*, *Ordering*, *Customers*, *Products*, etc.

A bold design technique is to introduce three global namespaces for these schemas, that is, ".../Common", ".../Request" and ".../Response". Such a design technique simplifies the work on WSDLs. By importing these global namespaces, the <types> sections of all the WSDLs become same within an enterprise web services system. That liberates developers from the configuration work on WSDLs. Figure 2 shows a code snippet of such a design.

```
<?xml version="1.0" encoding="UTF-8"?>
<definitions name="AccountService-v1.wsdl"
 targetNamespace=".../AccountService"
 xmlns:acct=".../AccountService"

 <types>
  <schema
   targetNamespace=".../AccountService"
    xmlns:acct=".../Account/...">
    <import namespace=".../Request/..."
     schemaLocation="./.../request.xsd"/>
    <import namespace=".../Response/..."
     schemaLocation="./.../response.xsd"/>
    <import namespace=".../Common/..."
     schemaLocation="./.../common.xsd"/>
  ... ...
  </schema>
 </types>
 ... ...
</definition>
```

Figure 2. Using proxy schemas

3) Use proxy schemas: So far so good. However most WSDL editors are not happy with such 'import', where multiple schemas are sharing a single namespace. Because WSDL 1.1 specification does not allow do so [3]. To make it work, proxy schemas are used.

A proxy schema is a schema that assembles multiple schemas (component schema) of a single target namespace [16]. The first benefit of proxy schema is simplifying WSDL, so that developers now focus on operations and bindings in WSDLs. Secondly a proxy schema also hides component schemas from WSDL, thus makes WSDL and service schemas loosely coupled. Knowing that, developers are able to update either WSDL or component schemas as long as no conflict occurred.

The 'proxy schemas' approach fully uses the robust import mechanism of WSDL, which allows the service schemas and WSDL are prepared by different development groups. In the case study, the service schemas were in fact developed prior to the development of web services. Using proxy schemas, the service schemas focus on the structure aspect of services, while WSDL concerns over the connectivity aspect of services. That achieves the principle

[3]WSDL 1.1 states that: WSDL allows associating a namespace with a document location using an import statement [13]. So WSDL editors do not allow for multiple imports with a same namespace. This is changed in WSDL 2.0 where it says 'If a WSDL 2.0 document contains more than one wsdl:import element information item for a given value of the namespace attribute information item, then they must provide different values for the location attribute information item' [15].

of Separation of Concerns or SoC.

C. Deploy versioning over services

There is no explicit mechanism of web services versioning in WSDL 1.1. Therefore whenever WSDLs or service schemas get changed, a new service with a new namespace has to be created. That is of course a drastic design. To solve this problem, our practice is to embed version information in web service contracts.

Particularly, versioning formats from [17] and minor-changes rules from [18], [19] are successfully adopted in the case study. Besides to these, we introduced some rules of version formats and service schema settings, and rules of delivering multiple minor-version services.

1) Major-minor version formats: A generic major-minor versioning approach of software development is also valid for web services, which accommodates two levels of changes. A major version change is a significant update that may bring incompatibilities, while a minor version change is such an update that is backward-compatible with existing systems. The following formats are used in WSDLs and service schemas.

- Place a number (n) as a major version in WSDL's targetNamespace, using '-v' as prefix. eg. <wsdl:definitions targetNamespace= "http://.../wsdl/AccountService-v1"
- Encode major and minor version (m.n) in the targetNamespaces of inline schemas within WSDL's <types> section, using '-v' as prefix. eg. <wsdl:types> <xs:schema targetNamespace= "http://.../wsdl/AccountService-v1.0">
- Attach a numeric date (yyyy/mm) as version number in the targetNamespace of an independent service schema. A schema of minor changes should use the same targetNamespace as that of major version, while stay at a different location. eg. <xs:schema targetNamespace= "http://.../xsd/Account/2008/12" (major version: "../2008/12/account.xsd") (minor version: "../2008/12/18/account.xsd")
- Embed major-minor version (m_n) in the names of services, bindings and portTypes, using '_v' as prefix. eg. <wsdl:portType name="AccountService_v1_0">

2) Rules of minor version changes: [18] and [19] have proposed practical rules of minor version changes, which greatly help achieve backwards compatibility of web services. The details of the rules are omitted for brevity.

3) Configuration of multiple minor-versions: Figure 3 is a code snippet showing how to apply the above formats and rules to configure multiple minor-version services in one WSDL, so that makes web services more flexible and agile.

In the snippet, the WSDL defines a major-version web service *AccountService-v1*, which includes two minor-version services: *AccountService-v1.0* and *AccountService-v1.1*. Of

```
<?xml version="1.0" encoding="UTF-8"?>
<definitions name="AccountService-v1.wsdl"
 targetNamespace=".../AccountService-v1"
 xmlns:v1_0=".../AccountService-v1.0"
 xmlns:v1_1=".../AccountService-v1.1" ...

 <types>
  <schema
   targetNamespace=".../AccountService-v1.0"
   xmlns:acct=".../Account/2008/12">
   <import namespace=".../Account/2008/12"
   schemaLocation="./2008/12/account.xsd"/>
  </schema>
  <schema
   targetNamespace=".../AccountService-v1.1"
   xmlns:acct=".../Account/2008/12">
   <import namespace=".../Account/2008/12"
   schemaLocation="./2008/12/18/account.xsd"/>
  </schema> ...
 </types>

 <message name="addContact">
  <part element="v1_0:addContact" .../>
 <message name="updateContact">
  <part element="v1_0:updateContact" .../>
 <message name="addAccountWithUrl">
  <part element="v1_1:addAccountWithUrl" .../>
 <message name="mailAccountDetails">
  <part element="v1_1:mailAccountDetails" .../>...

 <portTtype name="AccountService_v1_0">
  <operation name="addAccount" >...
  <operation name="updateAccount" >...
 <portTtype name="AccountService_v1_1">
  <operation name="addAccount" >...
  <operation name="updateAccount" >...
  <operation name="addAccountWithUrl" >...
  <operation name="mailAccountDetails" >...

 <binding name="AccountServiceBinding_v1_0"
  type="tns: AccountService_v1_0" >...
 <binding name="AccountServiceBinding_v1_1"
  type="tns: AccountService_v1_1" >...

 <service name="AccountService_v1_0"
  <port binding =
   "tns:AccountServiceBinding_v1_0" ..>
 </service>
 <service name="AccountService_v1_1"
  <port binding=
   "tns:AccountServiceBinding_v1_1" ..>
 </service>
</definitions>
```

Figure 3. Configuration of multiple minor-versions

them, *AccountService-v1.0* defines two operations, *addAccount* and *updateAccount*, and *AccountService-v1.1* adds two extra operations. The operation *mailAccountDetails* is a brand-new one, while *addAccountWithUrl* a sibling of *addAccount* with an extra parameter "Url". Owing to the design, all the operations of *AccountService-v1.0* are continuously valid in *AccountService-v1.1* even when *AccountService-v1.0* retires sometime later on.

D. Filters cooperating with contracts

Writing web service filters usually has to deal with two issues. One is to collaborate with service schemas, and the other to make filters be fine-grained at business entity level.

1) Cooperate with service schemas: Though the idea of filter existed in Apache Axis [2] and various filter frameworks in web services engines, the framework of Spring-WS [6] offers a same approach for writing interceptors and endpoints, both manipulate service schemas. Using the approach, four types of filters were developed in the case study: Validation filters, Logging filters, Transformation filters and Security filters.

2) Make filters fine-grained: The characteristics of enterprise web services require service policies be fine-grained at the level of business entities and transactions. For instance, it often requires have a fine-grained security control at the level of business entity. This request can be fulfilled by controlling the security information through web service messages, which is exactly the advantage of contract-first methods. Particularly, a design pattern of using Aspect-Oriented-Programming or AOP technique [20] is adopted to make the design work.

Figure 4 shows how it works. On the one hand, the filter *MySecurityInterceptor* inherits from superclass *SuperInterceptor* of the framework, which secures the authentication of access. On the other hand, the security policies are populated to business entity *AccountMgr* by plugging a handler *AOPProxy* in the *AccountMgr* so that control the security at the level of the entity *AccountMgrImpl* and its transactions.

IV. Related Work

The above design techniques are inspired by major contract-first methods, SOA methodologies and some design techniques of WS and XML schemas.

A. Contract-first methods

There are a number of contract-first methods. Each has its own special features. The followings are some typical ones.

1) Spring-WS: [6] gives developers full freedom to design the contracts of web services. Incorporating an improved contract-first method, Spring-WS claims that neither programming code nor WSDL is auto-generated by its engine, which significantly differentiates it from other contract-first methods.

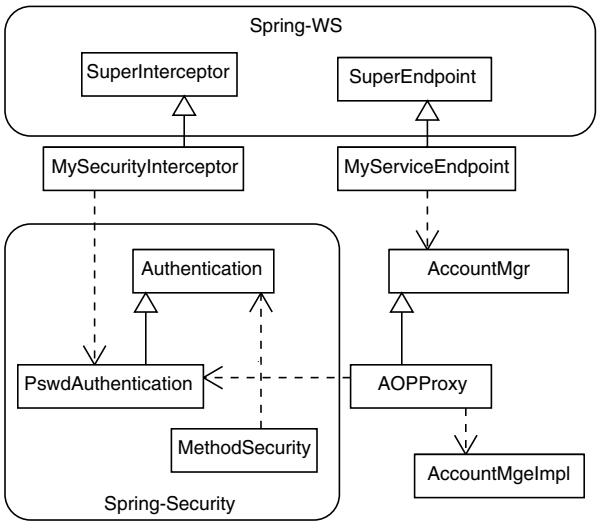

Figure 4. Collaborating business entities

2) Apache Axis2 and CXF: [7] and [8] describe how to implement contract-first web services with Apache Axis2 and Apache CXF. Both methods take an opposite direction of code-first, that is, to generate code from WSDL by means of the same engines as those of code-first. Doing so does not provide much room of designing web service contracts.

3) Web Services on .NET: [9]–[11] present various approaches for developing contract-first web services, especially using C#.NET on Visual Studio IDE. In the methods, all the tasks are closely working on Visual Studio IDE, and the service schemas and WSDLs are generated by tools. A good point of these methods is that the service schemas are smoothly associated with business entities.
However most of the generation tools do not have the ability of controlling versioning and namespaces, this brings a challenge to embed design techniques into the tools and to strengthen the discipline of designing service schemas and WSDLs.

B. SOA methodologies

SOA methodologies were also carefully examined during the course of the case study.

1) Artifact-Centric Modeling: Artifact-centric modeling from Bhattacharya et al [22] is a formal method of modeling business process. An interesting thing here is that it shifts the focus from actions to business entities, and uses them in the form of artifacts, as the driving force in the design of business processes. That is very similar to the situation of contract-first design techniques where the focus is shifted from functions to contracts, and service schemas are put at a central point of web services development.

2) Services Oriented Modeling and Architecture: Services Oriented Modeling and Architecture or SOMA is a full-blown modeling methodology by IBM [23]. A key point

of SOMA is to model, or analysis and design, services from the point view of services rather than objects or components. SOMA emphasizes comprehensive blueprint for design process. To do so, SOMA defines multiple aspects of services, including dependencies, composition and messages. Contract-first design techniques follow the same direction regarding the contracts of web services.

SOMA is a high level modeling and architecture methodology for enterprise SOA systems, while contract-first design techniques fulfill detailed methodologies of designing web service contracts to show practical applicability.

C. Design WSDL and XML Schema

There are also industry efforts and publications regarding the techniques of designing WSDL and XML schemas.

1) WS-I Profile: Focusing on web services interoperability, [24] or WS-I Profile defines a set of conformance rules of XML schemas and WSDls. Particularly, it identifies the rules for SOAP Messaging (SOAP Envelopes, SOAP Processing Model, SOAP Faults, Use of SOAP in HTTP), Service Description (WSDL and Schema Document Structure, Types, Messages, Port Types, SOAP Binding), UDDI registry data (Service Publication and Discovery) and Security (Use of HTTPS).

2) Design Patterns of XML Schema: On the other hand, [25] presents design patterns of XML schemas with considerations for WSDL design, including Russian Doll, Salami Slice, Venetian Blind, and the mixtures of these patterns. All of design patterns, as well as the WS-I conformance rules, are more helpful in detailed design of XML schemas, though they also provide great references to web service architecture.

V. Conclusion

Comparing with the above methods, the design techniques presented in the paper not only develop WSDL and service schemas firstly, but also provide detailed methodologies to guide developers go step by step to fulfill the designs of web service contracts.

Focusing on enterprise system architecture, the techniques have covered most of the key aspects of designing web service contracts, especially include consolidating service schemas in line with business entities, constructing coarse-grained namespaces, applying versioning over WSDLs and service schemas, and writing filters by collaborating with contracts. By applying the techniques to practice, not only does the development increase reusability and productivity, but also the web services improve agility and compatibility.

The proposed techniques are under refinement, also it is anticipated to formulate formal models for the techniques. In particular, some quantitative models are expected to consolidate services granularity by calculating the distances of service structures and behavior, to predicate service compatibility by quantifying the characteristics of versioning functions, and to numerically measure service complexity by evaluating the matrices of web services.

References

[1] W. Chou, *Web Services: Software-as-a-Service, Communication, and Beyond*, ICWS 2008 Keynote, available from http://conferences.computer.org/icws/ 2008/keynote1. pdf, pp. 10-13, Sep 23, 2008.

[2] Axis2, *Apache Axis2*, available from http://ws.apache. org/axis2/, August 2008.

[3] CXF, *Apache CXF: An Open Source Service Framework*, available from http://cxf.apache.org/, Oct 2008.

[4] S. Loughran and E. Smith, *"Rethinking the Java SOAP Stack"*, in proceedings of ICWS 2005, July 2005.

[5] T. Erl, *Service-Oriented Architecture: Concepts, Technology, and Design*, Upper Sadle River: Prentice Hall PTR, 2005.

[6] A. Poutsma, *Tutorial: Writing Contract-first Web Services*, available from http://blog.springframework. com/arjen/archives/2006/06/09/tutorial: writing contract-first web services/, June 9, 2006.

[7] A. Ranabahu, *Contract-First Web Services with Apache Axis2*, available from http://blog.springframework.com/arjen/archives/2006/ 06/09/tutorial-writing-contract-first-web-services/, Aug 8, 2006.

[8] C. Schneider, *Defining Contract first webservices with wsdl generation from java*, available from http://cwiki.apache.org/CXF20DOC/defining-contract-first-webservices-with-wsdl-generation-from-java.html, Aug 21, 2008.

[9] J. Hasan, *Expert Service-Oriented Architecture in C#: Using the Web Services Enhancements 2.0*, pp. 37-65, Apress, 2004.

[10] A. Skonnard, *Contract-First Service Development*, MSDN Magazine, available from http://msdn.microsoft.com/enus/magazine/cc163800. aspx, May 2005.

[11] C. Weyer, B. de Silva, *WSCF - Web Services Contract First*, available from http://www.thinktecture.com/ resourcearchive/tools-and-software/wscf, 2006.

[12] V. Andrikopoulos, el at, *Managing the Evolution of Service Specifications*, in CAiSE 2008, LNCS 5074, pp. 359-374, 2008.

[13] C. Erik, et al. (Eds.), *Web Services Description Language (WSDL) 1.1*, available from http://www.w3.org/TR/wsdl, 15 March 2001

[14] A. Layman, et al. (Eds.) *Namespaces in XML, World Wide Web Consortium*, available from http://www.w3.org/TR/REC-xml-names.

[15] C. Roberto, et al. (Eds.), *Web Services Description Language (WSDL) Version 2.0*, available from http://www.w3.org/TR/wsdl20/, 26 June 2007

[16] H. S. Thompson, et al. (Eds.), *XML Schema Part 1: Structures Second Edition*, available from http://www.w3.org/TR/xmlschema-1/, 28 October 2004

[17] A. Trenaman, *WSDL Versioning Best Practis*, available from http://blogs.iona.com/sos/20070410-WSDL-Versioning-Best-Practise.pdf, Apr 2007.

[18] R. Betuk, *Make minor backward-compatible changes to your Web service*, IBM DeveloperWorks White Paper, available from http://www.ibm.com/developerworks/webservices/library/ws-backward.html, 2004

[19] E. Mark, et al. *Moving forward with Web services backward compatibility*, IBM DeveloperWorks White Paper, available from http://www.ibm.com/developerworks/library/ws-soa-backcomp/index.html, May 2006

[20] G. Kiczales el at, *Aspect-Oriented Programming*, in proceedings of the European Conference on Object-Oriented Programming (ECOOP), Finland. Springer-Verlag LNCS 1241. June 1997.

[21] Sun Microsystems, *Core J2EE Patterns - Transfer Object*, available from http://java.sun.com/blueprints/corej2eepatterns/Patterns/TransferObject.html, 2001-2002.

[22] K. Bhattacharya el at, *Towards Formal Analysis of Artifact-Centric Business Process Models*, in BPM 2007, LNCS 4714, pp. 288-304, 2007.

[23] A. Arsanjani, *Service-Oriented Modelling and Architecture*, available from http://www-128.ibm.com/developerworks/webservices/library/ws-soa-design/, November 2004.

[24] B. Keith, et al. (Eds.) *Basic Profile Version 1.1*, April 2006. available from http://www.ws-i.org/Profiles/BasicProfile-1.1.html.

[25] S. Hu, *XML Schema considerations for WSDL design in conformation with WS-I Basic Profile*, Oct 2006. available from http://www.ibm.com/ developerworks/xml/library/ws-soa-xmlwsdl.html.