A DYNAMIC PROCUREMENT B2B SYSTEM USING
PRIMITIVE WEB SERVICES

Takashi KOSHIDA
Department of Information Engineering, Matsue National College of Technology
14-4 Nishi-Ikuma, Matsue, Shimane 690-8518 Japan
takashi@matsue-ct.ac.jp

Graduate School of Information Science, Nara Institute of Science and Technology
8916-5 Takayama, Ikoma, Nara 630-0192 Japan
takas-ko@is.naist.jp

ABSTRACT

Two factors limiting expansion in the use of web services are that users find it
difficult to identify the actual web service they need or want to use, and that users
have difficulty in understanding both the function of a web service and the
Corresponding input/output interface. Moreover, the composition of web services is an
issue. The current challenge is to develop easily accessible and usable web services.
To solve these problems, we propose the concept of primitive web services. We define
these as fundamental web services with a unified name, function, and input/output
interface, which can be used commonly for various activities in many business fields.
Thus, both precise detection and understanding of web services will be needed. We
have developed primitive web services and agents to comprise and control them, and
designed a dynamic procurement business-to-business (B2B) system as an application
example.


1. INTRODUCTION

In recent years, there has been much research on the utilization of web services and
the development of web service composition. However, the use of web services in
business has not spread as quickly as expected. We believe this has been due to three
problems:
(1) It is difficult for users to find useful and suitable web services
(2) Users cannot easily understand the functions of web services and how to
use them
(3) Users cannot easily develop a stub program for executing web services on
the user side.

UDDI [15], which unifies the registration and retrieval of web services, was
proposed to resolve the first problem. However, the web service retrieval method is
limited to keyword and code reference, and problems regarding the reliability of
registered information – for example, although a web service is registered with the
UDDI, it might not actually exist – have arisen. Concerning the second problem, with
WSDL [16] which provides interface information for web services, semantic supports
which describe the restrictions and explain input data using RDF [14], DAML-S [5],
or OWL-S [12] have been proposed. However, a problem remains in that semantic descriptions might be given with no consistency for all web services. We previously considered the third problem based on WSIF [3], and in Koshida and Uemura [8,9,10,11] proposed a means of detecting web services dynamically from the UDDI registry, analyzed the WSDL file, designed and implemented a stub-less web service dynamic invocation system independent of output data types, and verified the effectiveness of this approach. To expand the use of web services in various business fields, our goal is now to find solutions to the first and second problems, and the realization of effective web service composition is indispensable. We propose the concept of Primitive Web Services (PWS) as a technique to solve the first and second problems, and thus enable effective cooperation of web services synthetically. Then we designed a dynamic procurement system using primitive web services and agents.

This paper is organized as follows. Related work is described in Section 2. Primitive web services are proposed in Section 3, and Section 4 describes their implementation and the design of a dynamic procurement B2B system using primitive web services and agents. We conclude in Section 5.

2. RELATED WORK

In recent years, the use of ontology and agents in the detection and composition of web services has been proposed [6,13]. In Fukuta’s paper [6] regarding web service composition using DAML-S and JADE [4], a system which automatically generates the JADE program code for composing web services is proposed based on the process description on DAML-S. However, no method where agents compose web services, and any processing of the agents to obtain execution results, has been reported. To compose and apply several web services, it is important to understand how the required web service is detected and used, and how the next web service is composed given the previous execution result. However, these have not been described concretely. We have developed a technology for actually executing web services and to enable web service composition that can take into consideration execution results by using JADE agents.

Paolucci and others [13] have developed techniques to raise the retrieval accuracy of web services. These techniques enable the addition of ontology data based on a DAML-S description to a UDDI registry. Although this was effective for detecting the sample web service with ontology data, it is ineffective for detecting web services without ontology data. To put these techniques into practical use, it is necessary to create effective ontology data for all web services, but is not touching the policy about this point at all, as well as a method for executing detected web services. As we describe in Section 3, each web service is registered in a UDDI registry as a primitive web service with a unified name, meaning, function, and input/output interface. Furthermore, the inharmonious nature and ambiguity of conventional web services can be overcome by creating a table of these primitive web services and opening it to the public on the Internet. Therefore, we can easily detect the primitive web services from the UDDI registry through conventional retrieval by keyword.
3. PRIMITIVE WEB SERVICES

3.1 Background
In various business fields, the most important conditions for ensuring that a web service is used actively are that a user can easily get to and execute the required service. That is, it is important that the service provider offers easily accessible web services appropriate to user demands. For a user, though, the important thing is the information acquired through a web service, and not the web service itself. Unfortunately, in practice each company independently provides arbitrary web services that have an intrinsic interface and function. Consequently, users must understand the function, the input-and-output interface, and the input conditions for each web service they want to use. This places a heavy load on users in that a significant amount of time and effort may be required to find and execute web services.

3.2 Definition of the primitive web services
At present, a web service user must at least understand the function and input-and-output interface of a service. Therefore, sufficient time to investigate and understand the service is needed in advance. To mitigate user inconvenience, a fundamental web service in which the name, function, and input/output interface are unified into a meaning should be prepared for every minimum unit of a business process. Once users understand the function and input/output interface, they can combine these fundamental web services, and can then build and perform arbitrary business processes.

This approach will improve conventional optional web service development, allow the design of fundamental web services that are as flexible as possible, and we can deal any business process with fundamental web service’s combination. We call fundamental web services with this flexibility primitive web services, and define these as fundamental web services with a name, function, and input/output interface unified into a meaning which can be used in common in various business fields. For example, in a commercial transaction such as a goods purchase, each process for goods reference, a trust check, an order, an inventory check, a price estimate, a time-for-delivery check, delivery, and price settlement of accounts is considered a primitive web service. Moreover, a table of these primitive web services can be created and UDDI registry reference through keywords can be improved with such a table. A standard web service for each business process unit can also be developed by combining primitive web services. That is, a standard web service will be composed of several primitive web services, and a web service provider will develop the stub program for executing these primitive and standardized web services and supply this to widely dispersed users. As a result, users will be able to easily execute web services appropriate to their purposes.

3.3 Benefits of using primitive web services
When two or more companies produce products with the same specifications, a potential purchaser compares the product of each company, and follows a process to choose the product that most closely corresponds to their choice criteria. In such a case, a user (the purchaser) wants to compare and sort out products efficiently under the same conditions for all competing companies. For example, in a goods price estimate, the important consideration is how the estimate that provides the greatest benefit to the user can be identified, and whether the web service itself is the same as
that of the other company is not important. The content and quality of a service offered through a web service are important. Below, we explain the benefits of using primitive web services for both users and companies.

(1) User benefits
Since primitive web services with their names, functions, and input/output interfaces set to a unified meaning are defined for every business field, and for UDDI keyword retrieval, the detection of target web services becomes easy. Moreover, for a user, the directions and functional acquisition of web services become easier, and usability is greatly improved.

(2) Web service provider benefits
When companies offer and use primitive web services, the web service development cost of each company is reduced, and management resources become available which can be invested in the company’s core competence. Moreover, the standardization of primitive web services in various business fields makes detection and cooperation through a UDDI registry easier, and it becomes easier to apply web services. Thus, the use of web services should increase and an expansion of business opportunities can be expected.

4. DESIGN OF A DYNAMIC PROCUREMENT B2B SYSTEM
We have developed and deployed primitive web services based on the concept described in Section 3. Then we designed a dynamic procurement B2B system using these primitive web services and agents. We present about this system in detail.

4.1 Use-case
A business process generally consists of several sub-processes, and we developed a primitive web service for every sub-process of the procurement process. We assumed a goods procurement process involving beer brewers, a wholesaler, and retailers as a use-case where beer was the goods. The brewers, who manufacture the goods, provide a stock-management method and a goods-order-received method that belongs the same primitive web service. Furthermore, several credit research firms that investigate the credit of a company are also assumed, and a credit check web service that is the same primitive web service is provided.

4.2 Workflow in the Use-case
A series of goods procurement processes are executed by the brewers, wholesaler, and retailers. The workflow is shown in Figure 1.
(1) Delivery-of-goods demand:
A retailer requests a transaction and sends a demand for delivery of goods to a wholesaler using Servlet order interface (Figure 2).

(2) Trust check demand:
A wholesaler sends a demand for an investigation of the retailer's credit to a credit research firm.

(3) Trust check reply:
Using a credit check web service, a credit research firm executes a credit check of the retailer and responds to the wholesaler (Figure 3). An agent (CreditServiceAgent1) executes dynamically the getCredit method in the CreditCheck1 primitive web service (PWS).

(4) Inventory check and price estimate demand:
If there is no credit problem, the wholesaler sends requests for an inventory check and a price estimate to the brewers. We assume there are multiple brewers who manufacture beer with the same specifications and that the wholesaler simultaneously requests an inventory check and a price estimate from three brewers. An agent (StockServiceAgent7) executes dynamically the getStockdetails method in the GoodsService6 PWS.

(5) Inventory check and price estimate response:
Using the getStockdetails, all three brewers simultaneously perform a goods inventory check and price estimate, and then respond to the wholesaler.

(6) Goods selection and order:
The wholesaler compares the price estimate results from the three brewers through the agent (StockServiceAgent7), sorts out the brewers according to the choice criteria (although various criteria can be considered, we assume a single criterion of lowest price), and places an order for goods.

(7) Completion of a maker goods order received:
The brewer that receives the order transmits a completion message to the wholesaler after completing the order-received process using the getOrders method in the GoodsService6. The agent (StockServiceAgent7) executes
(8) **Completion of a goods order received:**
In response, the wholesaler transmits the completion message of a goods order received to the retailer.

--- Welcome to Matsui-ya Wholesaler ---
Please enter the Company name and Order

<table>
<thead>
<tr>
<th>Company name</th>
<th>Tiger.CO., LTD.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Phone Number</td>
<td>218-625-0841</td>
</tr>
<tr>
<td>Beer Type</td>
<td>draft</td>
</tr>
<tr>
<td>Order case</td>
<td>30</td>
</tr>
<tr>
<td>Delivery date</td>
<td>2005/01/15</td>
</tr>
</tbody>
</table>

These are wholesaler's information:
Wholesaler name is Matsui-ya.
Wholesaler ID is Id55038

Figure 2. Servlet order interface for retailers

--- Company Credit Check ---
Thomas Credit Research Firm CO., LTD.
Results of the CreditCheck1 Primitive WS (method: getCredit)

<table>
<thead>
<tr>
<th>Company name</th>
<th>Tiger.CO., LTD.</th>
</tr>
</thead>
<tbody>
<tr>
<td>President</td>
<td>Mohammad K. Ali</td>
</tr>
<tr>
<td>Address</td>
<td>203 N. Goodwin Ave., URBANA, IL 61801</td>
</tr>
<tr>
<td>Phone Number</td>
<td>218-625-0841</td>
</tr>
<tr>
<td>Credit</td>
<td>OK</td>
</tr>
</tbody>
</table>

Figure 3. Example of the credit check results for retailer

### 4.3 System architecture

(1) **About primitive web services**
In this use-case, there are a total of four servers (those of three brewers and one credit research firm) that deploy primitive web services. In each brewer's server, two methods are deployed: **“stock-management”** (getStockDetails) and **“goods order-received”** (getOrders) method belonging to the **“GoodsService6”** PWS. The **“stock-management”** method requires input data consisting of a wholesaler identification number, a wholesaler name, the desired beer type, the desired number of cases, and the desired delivery-of-goods day. Output data consisting of a reply company name, a specification beer type, the name of the lowest priced brand, a bar code, a unit price, the number of specification cases, and a total price is returned. The **“goods order-received”** method requires input data consisting of a wholesaler identification number, a wholesaler name, the desired beer type, the desired number of cases, and the desired delivery-of-goods day. The completion message for an order received is returned as output data. The actual deployment from a brewer’s point of view is shown in Figure 4.
Moreover, the “credit check” (getCredit) method belonging to the “CreditCheck1” PWS is deployed in the server of the credit research firm. The “credit check” requires a retailer (company) name and its telephone number as input data. As output data, a company name, a president name, the address, a telephone number, and a credit check result (i.e., OK or NG) are returned to the wholesaler (see Figure 3). This deployment situation is shown in Figure 5.

We developed these web services using Axis-1.0 [1] and deployed them on Tomcat-4.1.27 [2]. We used Microsoft Access2000 as a DBMS and each primitive web service had access to it using JDBC.

(2) About agents
Three agents are deployed in the wholesaler: the agent (StockServiceAgent7) that invokes dynamically a maker's primitive web service, the agent (CreditServiceAgent1) that invokes dynamically the primitive web service of the credit research firm, and the client agent (client1) that controls these two agents.
The agent *(StockServiceAgent7)* compares the price estimate from the three brewers, and determines which goods are available at the lowest price. This operation is shown in Figure 6.

![Agents that work at the wholesaler](image)

Figure 6. Agents that work at the wholesaler

We developed these agents using JADE-3.0. JADE is a framework for developing agents through Java program language based on FIPA [7] specifications. Using this system configuration, we designed a dynamic procurement B2B system based on the workflow described in Section 4.2.

### 4.4 Dynamic invocation for primitive web services

According to the workflow given in Section 4.2, we show the dynamic invocation mechanism of primitive web services.

**(1) A goods order received and a credit inquiry**

First, a wholesaler receives a goods order request from a new retailer, and performs a retailer credit inquiry using an agent *(CreditServiceAgent1)* under a client agent's *(client1)* control. The agent retrieves a UDDI registry using the “Company A” value as a `businessName` argument and the “CreditCheck1” value as a `serviceName` argument, and gets the WSDL file. Then the agent analyses an input/output data type, if data type is a complex type, it creates dynamically a JavaBeans class that represents the data type. And the `getCredit` method of a credit research firm is dynamically performed by this agent (Figure 7~9), and a response (Figure 3) giving the investigation result is sent to the wholesaler.
Input parameter setting for each method:
1) getCredit,
2) getStockdetails, 3) getOrders

Figure 7. Dynamic invocation workflow for primitive web services
Step1: Input Data Type Analysis

WSDL file

input data type decision

Primitive data type

Complex data type

Figure 8. Detailed workflow for Step1
(2) An inventory check and a price estimate

If the credit of the retailer is confirmed (Figure 3), the “stock-management” method (getStockdetails) of the three beer makers will be executed dynamically one by one using an agent (StockServiceAgent7) according to Figure 7~9. The agent retrieves a UDDI registry using “maker1”, “maker2” and “maker3” values as businessId arguments and the “GoodsService6” value as serviceName, and gets the WSDL files. Then the agent analyses an input/output data type and an access-point of web service, if data type is a complex type, it creates dynamically a JavaBeans class that represents the data type and executes dynamically the getStockdetails methods on three brewers one by one. The input data for the getStockdetails is set from Servlet order interface (Figure 2). This input data will be automatically set for the three makers' getStockdetails methods once it is specified. This agent will then display each maker's stock and price estimate result.

Next, the output results of the getStockdetails for the three makers are shown in the maker1, maker2, and maker3 areas of Figure 10.
(3) Goods selection and order

An agent (StockServiceAgent7) compares these results, selects the goods with the lowest price, and reports to the wholesaler. The sorting result is shown in the top part of Figure 11. It turns out that the “Blue Beer” of maker2, which is the least expensive, is chosen. If satisfied with the selection result, the wholesaler will place an order for goods to the selected maker using the agent (StockServiceAgent7). The agent invokes dynamically the getOrders method of the selected maker. In this case, a dynamic invocation mechanism for the getOrders is the same as a case for the getStockdetails (Figure 7–9). The maker that receives the order will execute an order-received process using the getOrders, and returns the completion message of an order received to the wholesaler. The result of the getOrders is shown in the lower part of Figure 11.
4.5 System summary

The dynamic procurement B2B system using primitive web services and agents was designed for the use-case. Since the “stock-management” (getStockdetails) method was developed as the method of a primitive web service, it became possible to simultaneously perform an inventory check and a price estimate for the three makers. The agent automatically selected the preferred goods from among the estimate results. In a conventional system, the web services corresponding to the processes of stock management, price estimate, and order received would exist individually in each company. Furthermore, goods sorting processing that compares the price estimate result for each company would also have to be performed separately, and the complexity and load that the user faces would both increase in proportion to the number of companies being compared. In this use-case assuming three makers, execution of the comparison and sorting of an inventory check and goods for all three makers is immediately carried out using primitive web services and the agent. We estimate that this would probably lower the necessary number of processing man-days by one-third or more compared to performing the separate web services individually for each of the three companies.

The credit check for retailers that the wholesaler performs in this use-case is a typical example of a primitive web service that can be applied in almost any business field. The development and deployment of many such general-purpose primitive web services will make web service use and cooperation much easier, and will therefore lead to new business opportunities within companies.

5. CONCLUSION

In this paper, we have shown how primitive web services can be used to enable precise detection of desired web services that can be performed smoothly using a UDDI registry. Based on this concept, we designed and develop specific primitive web services and agents to execute dynamically them and enable cooperation. These agents and primitive web services are combined to create a dynamic procurement B2B system. The use of primitive web services can overcome the lack of harmony and the ambiguity with respect to the names, functions, and input/output interfaces of
existing web services. Required web services become easier to detect through keyword retrieval in a UDDI registry, and execution also becomes easier. Now we have developed and implemented this dynamic procurement B2B system.

We have discussed an example of how primitive web services can be used in goods procurement. Various other business processes need to be analyzed so that appropriate general-purpose primitive web services can be developed. To do this, we will have to examine in detail

(1) the nature and contents of input/output data for each business process unit,
(2) the module size needed for a primitive web service, and
(3) which design techniques best enable widespread use and reuse.

The provision of security will be an important application issue. Moreover, using semantics, we would like to work on a general-purpose agent system.

REFERENCES

http://www.uddi.org/