Web Intelligence and Agent Systems: An international journal 3 (2005) 1–20 1
IOS Press

WSIPL: An XML scripting language for integrating web service data and applications

Eric Lo[a,*], David W. Cheung[a], C.Y. Ng[b] and Thomas Lee[b]
[a]*Department of Computer Science, The University of Hong Kong, Hong Kong*
[b]*Center for E-Commerce Infrastructure Development, Department of Computer Science, The University of Hong Kong, Hong Kong*

Abstract Data processing and integration on heterogeneous data sources often require intensive human resources on coding, but applications developed for this purpose are usually inflexible to fulfill dynamic business requirements. To-date, the Web Service paradigm standardizes the programmatic interfaces for application-to-application communication. It has gained a considerable momentum as a means of facilitate processing and integrating heterogeneous data sources. To take full advantage of Web Services, we proposed an approach in which all operations and data sources are exposed as Web Services. Based on that approach, a novel declarative scripting language called *WSIPL* (*Web Services Integration and Processing Language*) is designed to drive the data integration and processing tasks via Web Services in minimal programming efforts. The reference architecture of a WSIPL system and implementation issues are discussed. Industries deployed with this technology found that WSIPL can successfully enhance their data integration and processing systems with higher flexibility and efficiency.

Keywords Web services, data integration, XML, workflow

1. Introduction

Today Internet applications, such as electronic bookstores, Web portals and digital libraries, often require integrated access to multiple heterogeneous data sources on the Internet. For example, a company may want to enquire the total inventory level of a particular product stocked in different regions via the Internet, while each regional office maintains its own inventory database for its regional stock.

XML [2], a highly flexible format for data storage and exchange, quickly picks up the role for Internet-based data integration. Because of its universal, self-describing and platform-neutral properties, XML has emerged as the *de facto* standard for information interchange. Currently, most of the XML-based integration and processing tasks are done in an ad-hoc manner. Developing applications for this purpose using general-purpose programming languages (e.g., C/C++ and Java) involves intensive coding efforts on low-level data handling such as XML data parsing, transforming and merging. Besides, such applications are usually highly customized and hardly reusable. To fulfill the very dynamic business requirements at this Internet age, there is a strong demand on the technology that can make the integration tasks simple and reusable in order to save manpower in system development, deployment and operation.

*Corresponding author. E-mail: ecllo@cs.hku.hk.

Web Services (also called *E-Services*)[1] are the new technology expected to address the above needs. A Web Service is an autonomous software component that can be uniquely identified by a URI and dynamically interacts with other Web applications using open standards and protocols, such as XML [2], SOAP [3], WSDL [5] and UDDI [4]. The Web Service paradigm is not only known to be served for a new distributed paradigm for building reusable components in distributed applications, it is also announced to be the next wave of Internet-based business applications which can fulfill many different kinds of applications in a dynamic and proactive way [14]. One of the merits brought by Web Services is the revolution of traditional heterogeneous data processing and integration systems. As Web Services standardize the programmatic interfaces for application-to-application communication, by migrating existing data sources and processors (e.g., XML transformation processors) to Web Services, or by wrapping a thin layer of codes on existing operations to provide Web Service interfaces, the flexibility of data processing and integration tasks greatly increases [20]. Extensively examined the research challenges of data integration using Web Services. The authors have identified several requirements for building a data integrator. However, writing a program to integrate Web Services with general-purpose languages like Java could still be very labour-intensive and error-prone because this may introduce many coding efforts in the follow aspects:

- Interacting with Web Services.
- Parsing, transforming, and composing intermediate XML data throughout the process.
- Resolving the dependency of invoking Web Services and parallelizing the invocations of independent Web Services.

Therefore, the use of general-purpose languages to integrate and process Web Services oriented data and operations could be cost-ineffective, and sometimes impractical as business requirements vary from time to time and from business partners to business partners. This has driven us to research and develop an XML-based scripting language, called *WSIPL* (*Web Services Integration and Processing Language*), for the rapid and systematic application development that involves Web Services integration and processing.

The followings are the major benefits of developing applications using WSIPL:

- **High Flexibility:** As it is impractical to write procedural codes for different business requirements with same objective (data processing and integration), therefore a handy scripting language is needed. The benefit of developing applications using WSIPL is more obvious when users are allow to manipulating the intermediate XML data contents along the process. This is analog to the benefits brought by current server-side scripting for the Web. For example, writing HTML directly within a PHP file.
- **High Interoperability:** As Web Services standardize the interface for application integration, all data sources and applications in our framework are exposed as Web Services. Different operations, such as XSL transformations and database queries, could be called through a Web Service interface (WSDL). If a proprietary or legacy system needs to be handled, it could also be wrapped with a Web Service interface. This way, the caller can invoke all operations transparently via Web Services, which consume and supply data in the same protocol (SOAP) and in same format (XML). These platform-neutral and standardized operations can then be reused in many different applications. For instance, a customer history querying service can be reused by clients from Customer Relationship Management division and Sales division, in the same protocol (SOAP), by the same interface (WSDL) and on any platforms. This can gain higher interoperability than traditional (or XML-based) integration systems which the operation interfaces are arbitrary. In addition, the WSIPL engine that integrates and processes Web Services is also exposed as a Web Service. Therefore a deployed WSIPL system can call another WSIPL system, resulting a share of resources through "a network of WSIPL systems".
- **Parallel Processing:** WSIPL provides constructs to coordinate its execution. We have identified two categories of coordination mechanisms, (1) execution, and (2) parallelization. The execution control mechanisms include conventional support of conditional branching (e.g. if-then-else, switch-case) and iteration (e.g., for-each). For instance, an email alert Web Service is invoked only when the total stock is below a certain amount as reported by other Web Services. The parallelization mechanisms allow two independent Web Services to be executed concurrently. When multiple Web Services cooperate to deliver an application, their invocations can

[1] In this paper, we will use the terms Web Services, E-services and services interchangeably.

be coordinated in an efficient manner. If Web Service A supplies data to Web Service B, B depends on A; B can only be invoked after A finishes its execution. On the contrary, if two Web Services do not depend on each other, they can be executed in parallel to achieve higher efficiency.

Currently there are large number of research [13,18–20,22] and industrial work [5–8,12] that work on Web Services components. So far, none of them address the key benefits on data processing and integration as WSIPL does. This paper presents WSIPL, its reference architecture and implementation, to illustrate how simple and handy to integrate and process Web Services.

The rest of this paper is organized as follows. Section 2 outlines the WSIPL desiderata and constructs through a motivating example. Section 3 presents the reference architecture of a WSIPL system and the details on WSIPL implementation. Section 4 discusses our past work on WSIPL and their acceptance in industries. In Section 5, we discuss some related academic and industrial work and how WSIPL differentiates. Finally, Section 6 contains concluding remarks and future directions of WSIPL. In Appendix A, a full WSIPL script of the motivating example is included for reference. Appendix B presents another supplementary example to show WSIPL can also integrate several Web Services as a new Web Service.

2. WSIPL desiderata and syntax

In this section, we outline the design objectives and core features of WSIPL constructs via an example. This example illustrates the requirements for today's data processing and integration systems. It is based on our real-life experience reported from the industries that drove our design of WSIPL initially. [2] Interested readers can refer to the complete WSIPL script of the motivating example in Appendix A. Then, we introduce how the WSIPL language constructs can address the requirements raised by the application example easily. The formal language specifications are described in [10].

2.1. Motivating sxample and desiderata

An international company has two branches in United States and Hong Kong. It has to query the inventory of both branches for in-house logistic management everyday. The two query results need to be consolidated for querying the total inventory of a particular product. If the total inventory level falls below a user specific threshold, the management service should send warning emails to the managers-in-charge automatically. The inventory of each branch is supported by its own backend system, with different technologies deployed and platform. Now, from this motivating example, we can outline the desiderata for a Web Services based data integration and processing system:

- Accepting input parameters or queries from users, e.g., accepting user queries on a product inventory level.
- Transforming intermediate data contents, e.g., transforming the user input message to be the input message of the Web Service(s).
- Invoking Web Services, e.g., invoking the data query Web Services of Hong Kong and United State branch.
- Integrating data, e.g., consolidate the query results from both Web Services.
- Conditional branching, e.g., if the total quantity of an item in both inventories falls below a certain threshold, actions should be taken to alert the users.
- Exception handling, e.g., if there is any exception, the system can handle exception gracefully.

To fulfill the above desiderata, the inventory management application can be developed by using procedural languages such as C++ and Java, or it can be implemented as a part of the integration system for retrieving Web information. However, using these approaches need to handle network programming and data processing explicitly. Moreover, the application soon becomes one monolithic piece or several pieces of tightly coupled running codes, tied to a specific programming language or system platform. Whenever there is any logical change in data retrieval process, for instance, the schema of the Web Service changed, or the criteria to data integration changed based on the

[2] Please refer to the discussion in Section 4 for more details.

Table 1
WSIPL syntax notation

*	Zero or more occurrence
+	One or more occurrence
?	One or zero occurrence

business decisions, then the application, to a certain extent, has to be altered in programming level and redeployed in the system. As the business logic changes from time to time, such a repeated alteration to the application is time consuming and prone to error, especially without the aids of programming experts.

Therefore, the principle of our WSIPL is to tackle the above desiderata with minimum coding efforts. It is aimed to identify and model common tasks for integrating data by Web Services and provides easy-to-use language constructs to describe these tasks in an XML script file. Any change in business logic only alters the WSIPL script file and it is then automatically reloaded into our system architecture for execution. Based the criteria described above, we propose some important constructs of WSIPL as follows.

2.2. WSIPL script

A WSIPL script conceptually models a number of data processing and integration tasks. It is an XML file with the root element being **<wsipl:script>** ("`wsipl`" is the namespace prefix[3]). The notations used in this paper is in Table 1. The syntax of element **<wsipl:script>** is:

```
<wsipl:script version="1.1">
  <!-- <wsipl:source> -->
  <!-- <wsipl:variable>* -->
  <!-- <wsipl:task>+ -->
  <!-- <wsipl:response> -->
  <!-- <wsipl:exception-handler>* -->
</wsipl:script>
```

The *version* attribute indicates the version number of a WSIPL script. The child elements of **<wsipl:script>** will be explained in the following subsections. The order in which these child elements under **<wsipl:script>** occur is not a matter because WSIPL is declarative in nature, i.e., it specifies what tasks should be done instead of the tasks order.

2.3. Source message

As mentioned in the desiderata, WSIPL have to model user inputs. Thus, the element **<wsipl:source>** is defined for modeling the user inputs. Its syntax is: **<wsipl:source** *name*="qname"/>. It binds the input message to a name specified by *name* attribute with value is "qname". The following binds a name, IncomingQuery, to the input message received by the WSIPL system. The source message should be a message sent from a client and it contains a query about a product as in our example:

```
<wsipl:script version="1.1">
  <wsipl:source name="IncomingQuery"/>
  <!-- Other WSIPL constructs -->
</wsipl:script>
```

[3]The namespace declaration *xmlns:wsipl* ="http://www.csis.hku.hk/~wsipl" is omitted in this paper for ease of exposition.

2.4. Variable

Variables are required to hold data values along the data integration process. The syntax of a WSIPL variable is: **<wsipl:variable** *name*="qname"*select*=" string-expr"/>. A variable is a name that is bound to a value which is evaluated from an XPath [11] expression. The *name* attribute specifies the variable name and the *select* attribute specifies the XPath expression that generates the value of this variable. For the sake of simplicity for both language implementors and script users, the variable syntax and usage are identical to XSLT. For example, the following expression binds variable "SQL" to the result of XPath expression which concatenates the predefined SQL statement "SELECT Modelname, Quantity, Manager FROM Inventory WHERE Product =" with the product value extracted from the source message by the XPath expression "`/data/namevaluepair [@name, product]`", while the incoming message declared by **<wsipl:source>** serves as the input contents for variable:

```
<wsipl:script version="1.1">
  <wsipl:source name="IncomingQuery"/>
  <wsipl:variable name="SQL" select=concat(
   'SELECT Modelname, Quantity, Manager
   FROM Inventor WHERE Product=',
   /data/namevaluepair[@name, 'product'], ')'/>
  <!-- Other WSIPL constructs -->
</wsipl:script>
```

The variables can be referenced by placing a "$" symbol before the variable qualified name. For example, the variable "SQL" can be referenced by "$SQL". A variable is said to be instantiated when it is bound to a value for the first time. A variable can be defined in two different levels of scopes, where the variables in different scopes have different visibility within the whole script document.

2.4.1. Global variables

A variable that is nested directly under script element is a global variable. A global variable is visible to all WSIPL elements. The variable "SQL" before is a global variable.

2.4.2. Local variables

A variable that is instantiate under **<wsipl:task>** (Section 2.5) or **<wsipl: exception-handler>** (Section 2.9) is a local variable. The *select* attribute of a local variable takes the output of its preceding executed instruction as input. The scope of a local variable begins from the point where it is instantiated inside **<wsipl:task>** or **<wsipl:exception-handler>** until the closing of these two elements. If the local variable is instantiated in a nested block, it is also visible in the blocks containing this nested instruction block as long as all these instruction blocks are in the same **<wsipl:task>** or **<wsipl:exception-handler>** element.

2.4.3. Shadowing and changing values of variables

The value of a global variable cannot be changed but can be shadowed by local variables. When a global variable and a local variable have the same variable name, access to this variable name within the scope of the local variable will return the value of the local variable but outside this scope will return the value of the global variable.

The value of a local variable can be changed by reapplying **<wsipl: variable>**. The old value will be disposed and will be replaced by the new value. In other words, the language engine should maintain a store for global variable bindings and an independent store for each local variable scope.

2.5. Task

In data integration and processing, some execution steps often form a logical procedure like the procedure calls or methods in programming paradigms. Continue with the example, to invoke query service in US, the inventory management service should (i) prepare a message according to the target Web Service schema, (ii) send this message to the target Web Service, (iii) receive the returning message from the Web Service and extract the data from it. These steps are essentially the same as those in the invocation of the Web Service in Hong Kong. Moreover, the

two invocations of Web Services do not depend on each other and they can be executed concurrently. Element **<wsipl:task>** is defined to group some logical related execution steps into a task and its syntax is:

```
<wsipl:script version="1.1">
  <wsipl:source name="qname"/>
  <wsipl:variable name="qname" select="string-expr"/>
  <!-- <wsipl:variable>* -->
  <wsipl:task name="qname">
    <!-- <wsipl:initial-content> -->
    <!-- WSIPL instructions -->
  </wsipl:task>
  <!-- Other WSIPL constructs -->
</wsipl:script>
```

The *name* attribute of task element specifies the name of task. **<wsipl: initial-content>** element is an XML content manipulation element which compose a well-formed message input to be used within the task. After specifying the input to the task, each task can perform some instructions such as invoking Web Services and transforming XML data.

In our example, three tasks are outstanding:

- Querying the quantity of a particular product via HK branch database query service.
- Querying the quantity of a particular product via US branch database query service.
- Integrating the above query results and testing the inventory level, alert the users if the total inventory falls below a certain threshold.

Thus, the three outstanding tasks are modeled by three elements, namely **<wsipl:task** *name*="QueryUSInventory" **>**, **<wsipl:task** *name*= "QueryHKInventory"**>** and **<wsipl:task** *name*="InventoryLevelTesting">.

2.6. Manipulating XML in process

WSIPL offers two content manipulation elements for manipulate the XML contents throughout the whole execution process in the script. Previously, **<wsipl:initial-content>** is briefly introduced as an element nested inside a task for composing input contents for the whole task. Another element **<wsipl: compose>** serves a similar purpose on manipulating the intermediate contents within a task. Both content manipulation elements can nest with a content directive element **<wsipl:include>** for including the output of any tasks in the WSIPL script. The syntax is: **<wsipl:include** *name*=qname>. If a task includes another task output, it implies that the execution between two tasks is dependent. In our example, the task QueryUSInventory and QueryHKInventory are independent. Thus these tasks can be executed concurrently. Whereas the task InventoryLevelTesting depends on the query output of QueryUSInventory and QueryHKInventory, therefore InventoryLevelTesting must start after the two query tasks finished. This kind of parallelization is model implicitly by the **<wsipl:include>** element where a partial order tree is built in execution time. The system follows the order in the partial order tree to execute the tasks. The name attribute of **<wsipl:include>** element specifies which task output to be included. The followings show how the task InventoryLevelTesting includes the output of tasks QueryHKInventory and QueryUSInventory:

```
<wsipl:task name="InventoryLevelTesting">
  <wsipl:initial-content>
    <wsipl:include name="QueryHKInventory">
    <wsipl:include name="QueryUSInventory">
  </wsipl:initial-content>
  <!-- Invoke email alert if a particular
       stock falls below a threshold level -->
</wsipl:task>
```

2.7. Instructions

Each task should have a set of instructions in order to fulfill the task objective. There are two types of instructions: *Action* instructions and *Control* instructions. As Web Services standardize the function invoking procedure, we can model all actions by one **<wsipl:call>** element. The element **<wsipl:call>** abstracts the intrinsic tasks for sending messages through network, and it also simplifies the programming efforts on building a complete SOAP envelop. The syntax is:

```
<wsipl:call url="url" urn="urn" operation="qname"
   style="[RPC|DOC]" timeout="interval">
   <wsipl:param name="qname" type="type"
      value="value"/>
</wsipl:call>
```

In the task InventoryLevelTesting in our example, if a specific condition happens, an email alert Web Service should invoke. Intuitively, the email service should expect a set of inputs like email address and email subject. Thus another element **<wsipl:param>** is supported by WSIPL to model the input parameters for RPC [4] Web Services. The attributes *name*, *type* and *value* specify a parameter name, its data value type (e.g., string) and its data value respectively. On the other hand, the attributes *url*, *urn*, *operation* and *timeout* of **<wsipl:call>** specify the physical location, logical location, name of operation and the waiting time of the invoked service respectively. As defined in the WSDL 1.1 specifications [5], Web Services can be invoked by two different styles. Services in RPC-style (*style* attribute value is RPC) are designed to accept serialized representation of RPC method calls. In contrast, document-style services expect to accept SOAP messages carrying a generic XML document. To invoke a document-style service, the SOAP message contents can be constructed by an **<wsipl:compose>** element and specifies the *style* attribute of **<wsipl:call>** element with value DOC.

Now, the script for the task InventoryLevelTesting is:

```
<wsipl:task name="InventoryLevelTesting">
  <wsipl:initial-content>
    <wsipl:include name="QueryHKInventory">
    <wsipl:include name="QueryUSInventory">
  </wsipl:initial-content>
  <!-- If a testing condition is satisfied -->
   <wsipl:call url="http://company.com/soap"
   urn="urn:MailService"
   operation="sendAlertMail"style="RPC">
   <wsipl:param name="To" value="$manager"/>
   </wsipl:call>
</wsipl:task>
```

Each Web Service should have a WSDL file to describe its interface details. Thus it would be very easy for WSIPL users to know the input and output parameters of the invoked Web Services. The following XML snippet illustrates part of the integrated query result returned from HK and US database services.

```
<Modelname>Compact Disc</Modelname>
<Quantity>60</Quantity>
<Manager>cyng@hk.company.com</Manager>
<Modelname>Compact Disc</Modelname>
<Quantity>20</Quantity>
<Manager>ytlee@us.company.com</Manager>
```

[4] RPC stands for Remote Procedure Call.

2.8. Control instructions

Recall that in the desiderata, a scripting language designated for integrating data and Web Services must have a means to control the execution steps. WSIPL has four controlling instructions, namely **<wsipl:if>**, **<wsipl:choose>**, **<wsipl:for-each>**, and **<wsipl:throw-exception>**. The first three constructs share exactly the same usage as that in XSLT with the following syntax.

2.8.1. Conditional execution with IF

The **<wsipl:if>** instruction contains an instruction block, in which the instructions are executed when the boolean expression in the *test* attribute is evaluated to be 'true'. The syntax is:

```
<wsipl:if test=boolean-expr>
  <!-- WSIPL instructions -->
</wsipl:if>
```

Recall that the email alert service should be invoked if the total inventory level of a particular stock falls below a threshold. The **<wsipl:if>** element plays a key role in the conditional testing case. By writing the following script, the email alert web service is called only when the total quantity of two branches is lower than one hundred:

```
<wsipl:task name="InventoryLevelTesting">
  <wsipl:initial-content>
    <wsipl:include name="QueryHKInventory">
    <wsipl:include name="QueryUSInventory">
  </wsipl:initial-content>
  <wsipl:if test="number(//Quantity[position()=1]) +
    number(//Quantity[position()=2]) &lt; 100">
   <wsipl:call url="http://company.com/soap"
    urn="urn:MailService"
    operation="sendAlertMail" style="RPC">
      <wsipl:param name="To" value="$manager"/>
   </wsipl:call>
  </wsipl:if>
</wsipl:task>
```

2.8.2. Repetition with FOR-EACH

In our motivating example, if the total inventory level for a particular product falls below a threshold, the managers in charge of that particular product should be informed via email. This droves the design of the **<wsipl:for-each>** element. The **<wsipl:for-each>** element selects a node from the input contents using the XPath node-set expression in the *select* attribute and executes the instructions within the **<wsipl:for-each>** instruction block. The execution repeats for each selected node matching the node-set expression. A **<wsipl:loop-variable>** must be stated after the **<wsipl:for-each>** element and is used to bind a local variable to a value like **<wsipl:variable>** does. However, the string expression in the *select* attribute of **<wsipl:loop-variable>** is evaluated at the node selected by the **<wsipl:for-each>** instruction instead of the root node. The first instruction next to the last **<wsipl:loop-variable>** will use the root node of the initial contents of the **<wsipl:for-each>** instruction as its input contents. The syntax is:

```
<wsipl:for-each select=node-set-expr>
  <!-- WSIPL instructions -->
</wsipl:for-each>
<wsipl:loop-variable name=qname select=string-expr/>
```

Therefore, after checking the total inventory level (by **<wsipl:if>**) on the consolidated result(by **<wsipl:include>**), the task InventoryLevelTesting invokes the email Web Service for each email address nested in the element `Manager`:

```
<wsipl:task name="InventoryLevelTesting">
  <wsipl:initial-content>
    <wsipl:include name="QueryHKInventory">
    <wsipl:include name="QueryUSInventory">
  </wsipl:initial-content>
  <wsipl:if test=''number(//Quantity[position()=1])+
     number(//Quantity[position()=2]) &lt; 100">
   <wsipl:for-each select="//Manager">
    <wsipl:loop-variable name="manager" select="text()">
    <wsipl:call url="http://company.com/soap"
     urn="urn:MailService"
     operation="sendAlertMail" style="RPC">
      <wsipl:param name="To" value="$manager"/>
    </wsipl:call>
   </wsipl:for-each>
  </wsipl:if>
</wsipl:task>
```

2.8.3. Conditional execution with CHOOSE

<wsipl:choose>
Content: (wsipl:when+,wsipl:otherwise?)
</wsipl:choose>
<wsipl:when *test*=boolean-expr>
<!-- WSIPL instructions -->
</wsipl:when>
<wsipl:otherwise>
<!-- WSIPL instructions -->
</wsipl:otherwise>

The WSIPL includes the **<wsipl:choose>** instruction with the similar purpose as **<wsipl:if>**. It contains a list of **<wsipl:when>** and an optional **<wsipl: otherwise>** element. When an **<wsipl:choose>** instruction is executed, each of the **<wsipl:when>** elements is tested in turn. Only the instruction block of the first **<wsipl:when>** element whose *test* is "true" is executed. If no **<wsipl:when>** is true, the **<wsipl:otherwise>** instruction block will be executed. Due to limited space, the motivating example does not cover this element and more details are given in [10].

2.8.4. Exception throwing

<wsipl:throw-exception *code*=number *message*=string>

To fulfill the desiderata on handling exceptions, two elements **<wsipl:throw-exception>** and **<wsipl:exception-handler>** are designed. This two constructs model the traditional exception-trapping similar to Java, C++ and Delphi. The **<wsipl:throw-exception>** element throws an exception unconditionally. The *code* and *message* attributes specifies the exception code and message for pre-setting the variables in **<wsipl:exception-handler>**. This instruction can be used in the instruction blocks of both **<wsipl:task>** and **<wsipl:exception-handler>**. The exception will be handled by the corresponding element **<wsipl: exception-handler>**. Interested reader can also refer to [10].

2.9. Exception handling

An exception may occur when an instruction in a task or in a content directive fails or a **<wsipl:throw-exception>** intentionally throws an exception. The exception thrown can be handled by a **<wsipl:exception-handler>**. The *name* attribute in **<wsipl:exception-handler>** specifies the names of the **<wsipl:task>** elements from that the exceptions are handled. One **<wsipl: exception-handler>** may watch multiple tasks but a task can be watched by at most one **<wsipl:exception-handler>**:

<wsipl:exception-handler *name*=qnames>
```
  <!-- WSIPL instructions -->
```
</wsipl:exception-handler>

The **<wsipl:exception-handler>** element has an instruction block. The initial contents for this instruction block is the initial contents for the **<wsipl:task>** element that throws the exception. **<wsipl:exception-handler>** executes its instructions based on this initial contents. The final contents produced by this instruction block is used as the final contents for the **<wsipl:task>**. If an instruction in **<wsipl:exception-handler>** also throws an exception, the next instruction will not be executed and the **<wsipl:task>** that passes the control to this **<wsipl:exception-handler>** will throw an exception. It is as if no **<wsipl:exception-handler>** had ever handled this exception.

2.10. Returning result

After all tasks specified in the script finished, WSIPL offers a **<wsipl:response>** element to specify which task output should be regarded as the returned result. Its syntax is: **<wsipl:response** *name*=qname>. The *name* attribute binds the output of a task to the response message which is going to be received by clients. To complete our motivating example, the line below shows how to bind the output of the task InventoryLevelTesting to the response message:
<wsipl:response *name*="InventoryLevelTesting">

3. Implementing WSIPL

In this section, we first present the reference architecture of a WSIPL system for Web Services oriented data processing and integration. As Web Services interact by XML messages, the architecture can be adopted to all XML-based document processing systems like [15] and XML-based workflow systems like [23]. Section 3.2 presents our current implementation. A description on runtime operations of executing the example script is presented on Section 3.3.

3.1. WSIPL reference architecture

Figure 1 shows the various components of the reference architecture. The WSIPL system composed of a *WSIPL script repository*, a *SOAP gateway*, a *WSIPL engine*, a *pool of Web Services*, a *WSDL file* and a *Generic XML Schema*.
WSIPL Script Repository. The WSIPL script repository stores all WSIPL scripts. It can be in any fashion such as a local directory or an XML-database. Each script in the WSIPL script repository must have a corresponding entry in the WSDL file for describing the service it provides. The repository is also responsible for updating the WSDL files automatically. For example, whenever a WSIPL script is deleted from the repository, the corresponding entry in the WSDL file should be deleted automatically.
SOAP Gateway. The SOAP gateway acts as a communication interface of the system. It is in charge of the communication between the clients and different Web Services. Whenever a client invokes the WSIPL system by sending a SOAP request message, the SOAP gateway determines which service script the client has requested and notifies the engine to load the script from the WSIPL script repository. Meanwhile, the SOAP gateway extracts the

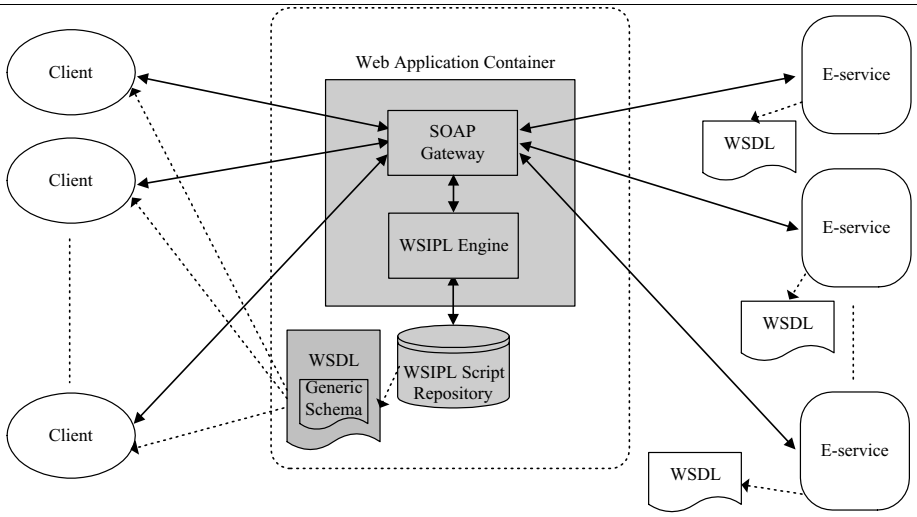

Fig. 1. Architecture of the WSIPL system.

payload from the client message and passes the payload to the WSIPL engine for execution. The SOAP gateway also acts as a client to invoke various Web Services.

WSIPL Engine. The WSIPL engine parses the service script and executes various instructions according to the logics in the script. When parsing the script, the engine resolves the dependency among the tasks and builds a partial-order tree. Based on the partial-order tree, the engine executes all independent service-oriented operations concurrently if possible. The service operations are invoked through the SOAP gateway. The engine is also responsible for validating the message payload extracted by the SOAP gateway to ensure that the client input messages conform to the service schemas. For the returning messages from different invoked Web Services, the engine acts as an integrator to integrate all the messages and passes the integrated result back to the clients via the SOAP gateway.

Pool of Web Services. The pool of Web Services contains the functional components and data sources. Recall that to facilitate data exchange, all operations should be exposed as Web Services. Some Web Service operations may require an access to resources such as databases and mail servers. Wrappers may be needed for those existing operations which do not have any Web Services interface, so that the WSIPL engine and the SOAP gateway can invoke all operations and third-party Web Services transparently. All services are invoked through the SOAP gateway.

WSDL File. A WSDL file is an XML document which describes all the details on how to invoke a Web Service. It defines the service location, types of input and output messages, transport protocols, etc. One WSDL file can describe more than one Web Services. Thus, WSDL file is stored in the system to describe all services the WSIPL system provides. Intuitively, a WSIPL script specifies a set of processing and integration tasks for a designated purpose. Therefore, each script can be identified as a single independent Web Service and should have a corresponding service entry in the WSDL file. The WSDL file can be stored in the local file system or publish to one or more public UDDI registries. Therefore WSIPL clients can refer to the WSDL file and customizes their own applications easily.

Generic XML Schema. As mentioned before, the WSIPL system generates different Web Service based integration and processing services by constructing different WSIPL scripts. Obviously, different services should have different sets of input parameters. Consider a WSIPL script that is designed to provide a weather forecast summary for four different cities. The script should invoke four weather forecast services and integrate the results into a single weather report. We expect the script to accept one input parameter (the number of days they want to forecast for the weather). On the other hand, another WSIPL script is designed to integrate two language translation services. Thus, a client can send a business contract in English to the WSIPL system and receive the contract in both Chinese and Spanish. For this instance, we expect the script to accept a document as input rather than a single data value. As there is no predefined set of input parameters for the WSIPL system, it is impossible to build a single, strongly-typed schema for describing all inputs. In WSIPL system, we have designed a generic schema that provides a standard interface for the WSIPL script users to refer to the input message. Thus, the payload of client SOAP messages should conform

```
<?xml version="1.0"?>
<xs:schema
 xmlns:xs="http://www.w3.org/2001/XMLSchema"
 xmlns:wsipl="http://www.csis.hku.hk/~wsipl"
 targetNamespace="http://www.csis.hku.hk/~wsipl"
 elementFormDefault="qualified">
 <xs:element name="data">
 <xs:complexType>
   <xs:choice minOccurs="1" maxOccurs="unbounded">
    <xs:element ref="wsipl:document"/>
    <xs:element ref="wsipl:namevaluepair"
      maxOccurs="unbounded"/>
   </xs:choice>
  </xs:complexType>
 </xs:element>
 <xs:element name="document">
  <xs:complexType>
   <xs:choice minOccurs="1" maxOccurs="unbounded">
    <xs:any namespace="##other"
            processContents="skip"/>
   </xs:choice>
  <xs:attribute name="name" type="xs:string"/>
  </xs:complexType>
 </xs:element>
 <xs:element name="namevaluepair">
  <xs:complexType>
   <xs:simpleContent>
    <xs:extension base="xs:string">
     <xs:attribute name="name"
       type="xs:string" use="required"/>
    </xs:extension>
   </xs:simpleContent>
  </xs:complexType>
 </xs:element>
</xs:schema>
```

Fig. 2. Generic XML Schema.

to the simple generic schema in Fig. 2. Typically, all incoming SOAP messages should be either in document style or RPC style. As defined in WSDL 1.1 specifications [5], the style of a call refers to whether a SOAP envelope is structured for sending XML documents or whether it is a serialized representation of an RPC method call. Thus the generic schema we proposed composes of two major elements, `<document>` and `<namevaluepair>`. The body of `<document>` is in the document style which has no constraints on how the document is structured. This kind of incoming messages imply that the corresponding WSIPL script understand the incoming XML document schema. Thus the contents of the incoming messages can be easily referenced by specifying an XPath expression beginning with `/data/document` in the WSIPL script. Another type of incoming messages is RPC based. The incoming parameters are represented as name-value pairs. Each `<namevaluepair>` element has an attribute “name” to specify the parameter name, and the contents of the element represent the parameter value. To refer to the input parameters in RPC style, the script can use the XPath expression `/data/namevaluepair[@name=`*qname*`]`.

```
<?xml version="1.0"?>
<xsl:stylesheet
 xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
 version = "1.0" xmlns:prefix1="URI1"
 xmlns:prefix2="URI2" xmlns:prefix3="URI3">
 <xsl:output omit-xml-declaration="yes"
             method="xml" indent="yes"/>
 <xsl:variable name="variable1" select="value1"/>
 <xsl:variable name="variable2" select="value2"/>
 <xsl:variable name="variable3" select="value3"/>
 <xsl:template match="/">
  <xsl:if test="boolean-expr">
   <true/>
  </xsl:if>
 </xsl:template>
</xsl:stylesheet>
```

Fig. 3. XSLT stylesheet for handling <wsipl:if>.

3.2. Implementation

We have implemented the WSIPL system as described above. All components have been implemented using JDK1.3. Both the SOAP gateway and WSIPL engine are implemented as Java Servlets, and the SOAP communication rides on HTTP. All functional units and data sources are exposed as Web Services with a WSDL file describing their properties. The XML parser is Xerces and the XSLT processor is Saxon. Currently, the engine and gateway are deployed on a Apache Tomcat servlet container and the script repository is a local directory.

Clients invoke WSIPL services by sending SOAP messages. The SOAP gateway then tells the engine which WSIPL script is requested by parsing the payload. Afterwards, the script is fetched from the script repository by the engine and execute various instructions specified in the script are executed.

As mentioned previously, control instructions (**<wsipl:if>**, **<wsipl: for-each>** and **<wsipl:choose>**) share the same syntax as that in XSLT. Thus the implementation of these constructs can be done by passing the conditional evaluation to an XSLT processor instead. Take **<wsipl:if>** as an example. In the runtime, if the WSIPL engine encounters this element, it will compose an XSL stylesheet as in Figure 3 (suitable namespace will be inserted if necessary). The XSL stylesheet will then be passed into an XSLT processor and the WSIPL engine will execute the instructions under **<wsipl: if>** element if and only if the result from XSLT processor is `<true/>`. The "variable1", "variable2", etc. and "value1", "value2", etc., are the variable bindings used in the Boolean expression (boolean-expr). The `<xsl:variable>` elements are used to set up the variable-binding environment for the expression evaluation.

3.3. Runtime operations

Now, we describe what happen during the motivating example WSIPL script is requested by a user and the engine executes it.[5] Figure 4 illustrates the steps in a user request as follows: (1) The client sends a SOAP request to the system, specifying which service script that he wants to invoke and the input parameters to that service (In this example, the input is the product's name). (2) The SOAP gateway extracts and passes the request payload to the WSIPL engine. (3) The WSIPL engine fetches the requested script from the script repository and executes the logic specified on the script. (4) There are three distinct tasks in example script: InventoryLevelTesting, QueryHKInventory and task QueryUSInventory. The first task includes the output from the last two tasks so that

[5]Complete WSIPL script of the motivating example is in Appendix A.

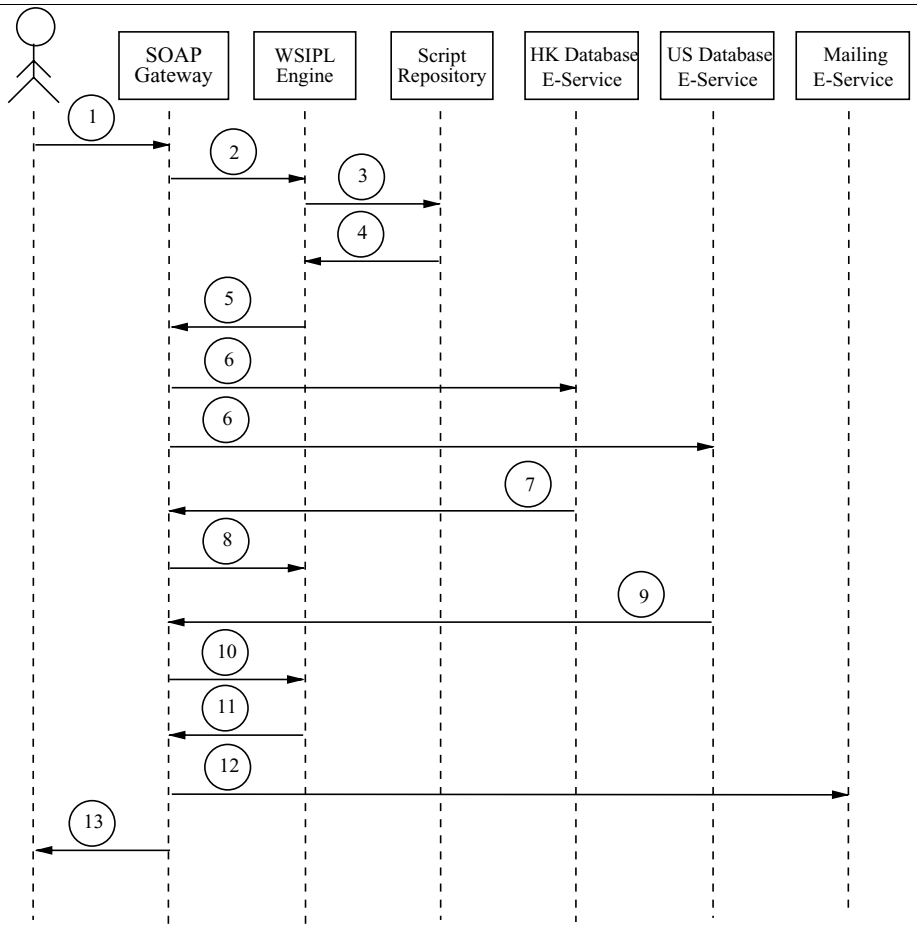

Fig. 4. Steps in inventory maintenance example.

the first task is dependent on the last two tasks. Thus the task InventoryLevelTesting can only be executed after the finish of the tasks QueryHKInventory and QueryUSInventory. (5) The engine passes the global variable SQL to the two database query services specified in tasks (QueryHKInventory and QueryUSInventory) via the SOAP gateway concurrently. (6–10) The SOAP gateway then collects the results from the two database services and passes the results to the engine. The engine now executes the task InventoryLevelTesting and checks if the total inventory from the integrated result is less than the threshold value. (11, 12) If it is true, the engine invokes the mailing service to send an alert email to every manager specified on the script. (13) Finally, the engine sends the integrated result back to the client.

4. Discussion

WSIPL has been developed in evolutionary an manner and is the third generation of this technology. The first generation is Document Integrator (DI) [15], which was designed in 1999 to meet the business requirements from an airfreight forwarder. The airfreight forwarder required a programmable XML-based middleware to integrate distributed and heterogeneous data sources from various business partners, such as warehouse operators and airlines, for its customers to obtain real-time logistics information. Despite having a limited instruction set, DI remains as the core integration framework being used through the three generations.

The second generation is XML Integrator (XI) and was designed in 2000. The programming language for XI (XIL) has incorporated a rich instruction set similar to the one proposed in WSIPL. The instruction set (e.g., variables,

control statements, and exception handling mechanisms) was specially designed to meet the Enterprise Application Integration (EAI) requirements from various business users. Several companies are currently using XI to integrate various procurement, accounting and sales systems to provide just-in-time decision support systems.

However, the Web Services concept and technology have not yet been materialized until now. With the wide acceptance of Web Services, the data exchange mechanisms between the integrator and various distributed data processors can be standardized so that cross-enterprise application integration becomes possible. The main difference between XIL and WSIPL is that the external adaptors (or transformation modules in [15]) that translate proprietary data formats to XML in XIL are excluded in WSIPL while all data processors are assumed to provide services via Web Services protocols. This way, the WSIPL framework has become less proprietary, more simple and interoperable with other Web Services technology.

Since the performance of our work is solely depends on the implementation skills but not the theory or concepts behind, we regard feedbacks from WSIPL users as a more important and usefulness accessment to this work.

Since 2002, the WSIPL system has been successfully deployed on several companies in Hong Kong [16]. According to our client feedbacks, they have written more than thousands WSIPL scripts to build up different supply chain management systems. According to the results of a short questionaire to clients recently, we found that all our clients can handle all the additional new integration tasks by their own non-technical teams. This result is encouraging because this exactly meet the flexiblity design goal of WSIPL.

In terms of interoperablity, all the clients report that the system can be fully integrated to all their existing systems around two months. They also report that the integration time period is relatively short because their operation systems need to use more than half year to be fully functional.

In terms of efficiency, we could only report the results from the client that motivates this project. It is because the rest of the clients has integrated the engine with many dependable systems (e.g., some manufacturing systems) already. For efficiency of the core system, our frieght forwarder partner reports that the system is highly efficient. So far, they grade the system as “no significant overhead” and “scalable”.

5. Related work

Our specification language and Web Services integration architecture are related to a number of academic and industrial work.

Industrial Standardization. Recently, industry has drawn much attention on the composition of various business applications into a business process. Business processes are explicitly driven by some Web Services composition language. In May 2003, a more complete specification on Web Services composition was proposed by IBM and Microsoft known as Business Process Execution Language for Web Services (BPEL4WS) [1]. It provides constructs for creating business processes by incorporating functions like correlation, fault handling, compensation, etc. A business process is a choreography of collaborating Web Services performed by different organizations. For example, a buyer invokes a *SubmitPO* Web Service to send a *purchase order* document to a seller. The seller invokes a *ProcessPO* Web Service to process the *purchase order*. After the *purchase order* is approved, the seller invokes a *ShipOrder* Web Service to instruct the physical delivery of ordered goods. After the buyer receives the goods, the buyer invokes a *ProcessPayment* Web Service to issue payment. An XML file written in Business Process Execution Language (BPEL) can model both executable business process and abstract business protocol. Web Services participated in the process then work according to the abstract flow logic specified in BPEL. One of the ultimate goals of BPEL is “Define-once, implement everywhere”. That is, after the business planner creates a set of business processes by BPEL, any partners (Web Services) that fulfill the business goals can also be participated in the process. In order to separate the application logic from the abstract business process, BPEL therefore does not specify the underlying implementation detail. In contrast, WSIPL can be identified as a declarative language, specifying the relationship among a predefined set of collaborating Web Services in implementation level. Since it is declarative in nature, the execution engine will execute the process by the best execution plan. We had done many works on the execution engine. However, since the details of the execution engine is out of the scope of this paper, and our implementation is deployed as part of the core system of our partner that motivates this project, we have to left out the details on this issue.

In addition, it has the flexibility that users can manipulate the intermediate XML contents directly on the scripts. Moreover, WSIPL models the integration tasks by its constructs explicitly. Whenever business requirements change, the corresponding WSIPL scripts can be altered to reflect the changes easily and directly. Altered scripts are reloaded into the integrator automatically and hot-deployment can be achieved. In additional, unlike BPEL, which is usually for specifying long-live process, WSIPL focus on real-time integration, that taking efficiency into account.

As a W3C note at the time of writing, XML Pipeline Definition Language (XPDL) [12] is a set of XML vocabularies for describing the processing relationships between XML resources. XPDL and WSIPL share similar design motivations and concepts. However, WSIPL has been evolving since its ancestor XML Integrator Language (XIL) and was proposed and implemented in [15]. Based on our experience in applying WSIPL and its predecessors, we have incorporated a much richer language constructs in WSIPL. XPDL is similar to XIL, the first generation WSIPL, which does not have any instructions for control statements (FOR-EACH, IF, CHOOSE), data and variable manipulations, and exceptional handling. Those instructions have been proved to be essential for integrating Web Services for industrial applications. Moreover, XPDL does not propose Web Services based implementation framework.

Programming Language. Integrating Web Services can be done by procedural languages such as C++ and Java, or it can be implemented by some development toolkits like WSDK [9]. However, we have to handle network programming and data processing explicitly if we use these approaches. Recently, XL [18,19](XML Programming Language) is proposed for for implementing Web Services. XL is a procedural language designed to operate on XML data in order to deliver Web Services. It is designed for the implementation of Web Services in general purpose. In contrast, WSIPL is a *declarative* scripting language to specify the relations of collaborating Web Services and processing them accordingly. The flexibility of WSIPL allows users to manipulate the intermediate XML contents directly in scripts and reduces the coding efforts. In addition, WSIPL allows parallel invocation of Web Services, but XL only supports sequential execution. While XL and WSIPL target for different application areas, WSIPL is a language specification that is fully implemented and deployed in industry. On the other hand, the implementation on XL is still ongoing at the time of this paper writing.

Others. Web Services based integration has been addressed in [20]. As mentioned in [20], Web Services can make the development of heterogeneous integration systems faster and less expensive to develop. They present an XML aggregator and a use case on how Web Services can be used to unlock heterogeneous business systems to extract and integrate business data. However, they have little coverage on how to drive their data integration engines as our WSIPL works on.

SELF-SERV [13] also presents a decentralized system on the composition of Web Services using statecharts [21], data flow and conversion rules; yet they propose a workflow model similar to BPEL where WSIPL performs real-time data integration for Web Services. Further, Web Services in their execution model communicate through Java sockets whereas we use SOAP which is known to be a widely adopted protocol for Web Services communications. Some other research [17,24] propose algebraic and algorithmic approaches on XML data integration and merge. However, they do not focus on the both processing and integration on heterogeneous data as we have done. Our work also related to some XML mediators such as Mentor [23]. Though their system architectures have some similarities with our reference architecture, they have little coverage on the design of a specialized language for driving the data as [20] and focus more on the “flow” like BPEL rather than the “integration and processing” like WSIPL.

6. Conclusions and future work

Data processing and integration of heterogeneous data sources often require intensive coding. However, applications developed for this purpose spend most of the efforts on low-level data handling and network programming where the end products are usually highly customized. Such applications cannot achieve their goals efficiently under the dynamic business requirements in the Internet age.

In this paper, we have identified the challenges for applying the Web Service paradigm on data processing and integration. We described a scripting language WSIPL which aimed at simplifying the intricate tasks on processing and integrating Web Services oriented operations. WSIPL provides constructs for Web Service invocations and control constructs to coordinate the processing and integration tasks. All independent tasks are executed in parallel to

attain high efficiency. WSIPL users no longer worry about the details of network programming, XML transformation, database connection mechanisms and Web Service invocation steps. The proposed language is conformed to all standards and the full WSIPL language specification can be obtained in [10]. We also present a complete framework for implementing WSIPL. The generic schema in the WSIPL system provides a powerful means for WSIPL scripts users to specify the input parameters for new services created by WSIPL script in a standardize way. Further, the usage of the generic schema can be extended and adopted to be a standard in referring to any kinds of XML incoming messages without schema known *a priori*.

Industries deployed with WSIPL found that WSIPL can successfully solve the data integration and processing tasks efficiently. User efforts are reduced and WSIPL system can attain different functionalities by writing different WSIPL scripts. Future work includes adding constructs to support query on the UDDI registry. Other data integration solutions such as schema matching and query discovery are now under development and will be integrated into next version of WSIPL. In long term, WSIPL should evolve into a standard for Web Services oriented data processing and integration.

References

[1] *Business Process Execution Language for Web Services*, Version 1.1, http://www-106.ibm.com/developerworks/webservices/library/ws-bpel/.
[2] *Extensible Markup Language* (*XML*), http://www.w3c.org/XML.
[3] *Simple Object Access Protocol*, http://www.w3c.org/TR/2001/WD-soap12-part0-20011217.
[4] *Universal Description, Discovery, and Integration*, http://www.uddi.org.
[5] *Web Services Description Language*, http://www.w3.org/TR/wsdl.
[6] *Web Services Experience Language*, http://www-106.ibm.com/developerworks/webservices/library/ws-wsxl.
[7] *Web Services Flow Language* (*WSFL 1.0*), http://www-4.ibm.com/software/solutions/webservices/pdf/WSFL.pdf.
[8] *Web Services Inspection Language*, http://www-106.ibm.com/developerworks/webservices/library/ws-wsilspec.html.
[9] *WebSphere SDK for Web Services* (*WSDK*), http://www.ibm.com/developerworks/webservices.
[10] *WSIPL specification version 1.1*, http://www.cecid.hku.hk/˜wsipl.
[11] *XML Path Language* (*XPath*) *Version 1.0*, http://www.w3.org/TR/xpath.
[12] *XML Pipeline Definition Language Version 1.0*, http://www.w3.org/TR/xml-pipeline.
[13] B. Benatallah, M. Dumas, Q.Z. Sheng and A.H. Ngu, *Declarative composition and peer-to-peer provisioning of dynamic Web Services*, In Proceedings of ICDE, February 2002.
[14] F. Casati, M.-C. Shan and D. Georgakopoulos, Guest editorial, *The VLDB Journal* **10**(1) (2001), 1–1.
[15] D. Cheung, S. Lee, T. Lee, W. Song and C. Tan, *Distributed and scalable XML document processing architecture for e-commerce system*, In Second Int. Workshop on Advanced Issues of E-commerce and Web-base Information System, June 2000.
[16] D. Cheung, E. Lo, C. Ng and T. Lee, *Web services oriented data processing and integration*, In Proceedings of the Twelfth International World Wide Web Conference, May 2003.
[17] V. Christophides, S. Cluet and J. Siméon, *On wrapping query languages and efficient XML integration*, (Vol. 29), In Proc. of ACM SIGMOD, May 16–18 Dallas, Texas, USA, 2000, 141–152,
[18] D. Florescu, A. Grunhagen, D. Kossmann and S. Rost, *XL: A platform for Web Services*, In Proceedings of ACM SIGMOD, June 2002.
[19] D. Florescu and D. Kossmann, *An XML programming language for Web Service specification and composition*, Bulletin of the IEEE Computer Society Technical Committee on Data Engineering, 2001.
[20] M. Hansen, S. Madnick and M. Siegel, *Data integration using Web Services*, In Int. Workshop on Data Integration over the Web, May 2002.
[21] D. Harel and A. Naamad, The statemate semantics of statecharts, *ACM Transactions on Software Engineering and Methodology* **5**(4) (1996), 293–333.
[22] M. Keidi, S. Seltzsam, K. Stocker and A. Kemper, *ServiceGlobe: Distributing e-services across the internet*, In Proceedings of VLDB, August 20–23, 2002.
[23] G. Shegalov, M. Gillmann and G. Weikum, XML-enabled workflow management for e-services across heterogeneous platforms, *The VLDB Journal* **10**(1) (2001), 91–103.
[24] K. Tufte and D. Maier, Aggregation and accumulation of XML data, *Bulletin of the IEEE Computer Society Technical Committee on Data Engineering* **24**(2) (20010, 34–39.

Appendix A: WSIPL in motivating example

Figure 5 depicted the WSIPL in the motivating example. This example script is for demonstrating the core usage of WSIPL only. Formal specification can be referred to [10].

```
<?xml version="1.0" encoding="UTF-8"?>
<wsipl:script
 xmlns:wsipl="http://www.cecid.hku.hk/~wsipl">
<!--Bind a name to the incoming query-->
 <wsipl:source name="IncomingQuery"/>
<!--Instantiate a variable-->
 <wsipl:variable name="SQL" select="
 concat('SELECT Modelname, Quantity, Manager
 FROM Inventory WHERE Product="',
 /data/namevaluepair[@name='product'], '"')"/>
<!--Query the HK Branch Database Web Service-->
 <wsipl:task name="QueryHKInventory">
  <wsipl:call url="http://hk.company.com/soap"
   urn="urn:DBWS" operation="query" style="RPC">
    <wsipl:param name="username" value="HKuser"/>
    <wsipl:param name="password" value="HKsecret"/>
    <wsipl:param name="SQL" value="$SQL"/>
  </wsipl:call>
 </wsipl:task>
<!--Query the US Branch Database Web Service-->
 <wsipl:task name="QueryUSInventory">
  <wsipl:call url="http://us.company.com/soap"
   urn="urn:DBWS" operation="query" style="RPC">
    <wsipl:param name="username" value="USUser"/>
    <wsipl:param name="password" value="USSecret"/>
   <wsipl:param name="SQL" value="$SQL"/>
  </wsipl:call>
 </wsipl:task>
<!--Integrate results and test inventory level-->
 <wsipl:task name="InventoryLevelTesting">
  <wsipl:initial-content>
    <wsipl:include name="QueryHKInventory"/>
    <wsipl:include name="QueryUSInventory"/>
  </wsipl:initial-content>
<!--If total inventory < 100,-->
  <wsipl:if test="number(//Quantity[position()=1])
    + number(//Quantity[position()=2]) &lt; 100">
   <wsipl:for-each select="//Manager">
    <wsipl:loop-variable name="manager"
      select="text()">
    <wsipl:call url="http://company.com/soap">
      urn="urn:MailingServices"
      operation="sendAlertMail" style="RPC">
      <wsipl:param name="To" value="$manager"/>
    </wsipl:call>
   </wsipl:for-each>
  </wsipl:if>
 </wsipl:task>
<!--Bind "Result" output to response message-->
 <wsipl:response name="InventoryLevelTesting"/>
</wsipl:script>
```

Fig. 5. WSIPL for motivating example.

Appendix B: An Integrated Currency Enquiry Web Service

This example show how a WSIPL system can be deployed as a "*service generator*". Figure 6 shows a WSIPL script that trying to integrate three independent currency enquiry services provided by three different banks. Each

```
<?xml version="1.0" encoding="UTF-8"?>
<wsipl:script>
 <wsipl:source name="USdollars"/>

 <wsipl:variable name="USdollars"
 select="/data/namevaluepair[@name='dollars']"/>

<!--//Input: US dollar; Output: HK dollar//-->
 <wsipl:task name="US2HK">
 <wsipl:call url="http://hkbank.com/service"
  urn="urn:CurrencyServices" operation="exchange"
  style="RPC" timeout="600">
  <wsipl:param name="dollars" value="USdollars"/>
 </wsipl:call>
 </wsipl:task>

 <!--//Input: US dollar; Output: Euro//-->
 <wsipl:task name="US2Euro">
  <wsipl:call url="http://eurobank.com/service"
  urn="urn:CurrencyServices" operation="exchange"
  style="RPC" timeout="600">
  <wsipl:param name="dollars" value="USdollars"/>
 </wsipl:call>
 </wsipl:task>

<!--//Input: US dollar; Output: Japan Yen//-->
 <wsipl:task name="US2Yen">
 <wsipl:call url="http://jpbank.com/service"
  urn="urn:CurrencyServices" operation="exchange"
  style="RPC" timeout="600">
  <wsipl:param name="dollars" value="USdollars"/>
 </wsipl:call>
 </wsipl:task>

<!--//Integrate the three results//-->
 <wsipl:task name="XMLtoHTML">
 <wsipl:initial-content>
  <wsipl:include name="US2HK"/>
  <wsipl:include name="US2Euro"/>
  <wsipl:include name="US2Yen"/>
 </wsipl:initial-content>
<!--//Transfrom the result to HTML//-->
  <wsipl:call url="http://servicegenerator.com"
   urn="urn:currencyXSLTtransform"
   style="DOC" timeout="600"/>
 </wsipl:task>

<!--//Return HTML output client//-->
 <wsipl:response name="XMLtoHTML"/>
</wsipl:script>
```

Fig. 6. Integrated currency enquiry service.

bank Web Service accepts a number in US dollars, and output the number in their local currency respectively. For example, a client pass the value “10” to the Web Service provided by hkbank.com, the Web Service will return

"78" to the client.[6] As there are no dependency between the Web Services invocation tasks, thus the US2HK task, US2Euro task and US2Yen task are executed in parallel. The task XMLtoHTML integrates the three results from the banking Web Services and pass the result to a XSLT transformation Web Services. The XSLT Web Services then transform the integrated result to HTML format and return to the client. The integrated service is identical to a new Web Service, where clients connect to that new service can enquire three currencies in one time. The integrated service can define custom method for clients to access. Clients need not examine every service interface, and need not handle various low-level programming to compose an integrated service. This simple example demonstrates how the WSIPL system can "generate" new services by integrating different Web Services in a cooperative way.

[6] 1 US dollar = 7.8 Hong Kong dollar.