Uppsala Master's Thesis in

Computer Science 308

2007-02-26

ISSN 1100-1836

Automatic Web Service Generator for Data Access

Feng Luan

Information Technology
Computing Science Department
Uppsala University
Box 337
S-751 05 Uppsala
Sweden

Supervisor: Manivasakan Sabesan
Examiner: Tore Risch

Abstract

Web services provide software components for application to application communication. This includes several public standards: WSDL provides a description of web service interfaces; SOAP is an XML style message protocol over underlying protocol, and XML Schema provides a type system for operation signatures. Using those standards, WSMOS (Web Service for Amos II) explores a mechanism which can deploy specified Amos II functions as a set of operations of a web service over the Internet. This report describes automatic generation of WSDL documents given Amos II functions to export, and immediate deployment of the exported functions. The WSDL generator binds exported Amos II functions to operations in a WSDL document, and specifies HTTP transport protocol and RPC/encoding style to be used. The WSMOS web server receives SOAP messages, parses their contents, calls Amos II functions, and sends back the SOAP messages of the results.

Table of contents

1. Introduction

A web service [15] is a software system for providing application to application communication via the World Wide Web. A web service is basically a web server which implements a set of *operations*. An application can retrieve data from a web server by calling operations.

Web Service Description Language (WSDL) [8] is an XML-based language for describing the operations of a web service. It describes the structure of arguments and results of each exported operation. Given a WSDL file, the customer of the web service will know the signature of each operation described by the file and how to invoke it.

Web service communication between the client and server uses the SOAP [7] message passing protocol. It is based on exchanging XML documents between clients and servers. Each SOAP message is represented as an *envelope* containing a *header* and *body* elements described with XML syntax. SOAP is independent of the used programming languages and platforms. Applications using different programming languages or platforms can communicate and access data using SOAP.

Amos II [1] is an object-oriented and functional database system. Amos II supports the mediator/wrapper approach [2] to query and update different and distributed data sources. Database users can query the database using an object-oriented SQL dialect, AmosQL [1]. The three basic concepts of the Amos II data model are *objects*, *types*, and *functions*. Types classify objects and each object is an instance of one or several types. Functions provide semantics of types. Amos II provides a programming interface for Java, C/C++, and Lisp.

The purpose of this project is to design a mechanism which can dynamically deploy any Amos II function as a web service operation without restarting the server and without developing or deploying any server Java code. The signature of each deployed Amos II function is described as a web service operation in an automatically generated WSDL document. The WSDL document thus describes the interface of the exported functions. The WSDL file is automatically generated, given names of exported Amos II functions, and the functions are immediately available as web service operations. An operation wrapping an exported function receives function arguments from the client and sends back the result of the function invocation as a collection. The web server does not need to restart when exporting and publishing new functions.

This report is divided into seven chapters. The first five chapters will introduce Amos II, XML and XML Schema, SOAP, and WSDL. These backgrounds are essential to understand this project. The implementation of this project is described in Chapter 6. It is divided to two parts. The first part describes how the WSDL document of exported functions is generated and the second is the functioning of the SOAP server. The last chapter concludes and suggests future works.

2. Amos II.

In this chapter, we will briefly introduce Amos II [1], which stands for Active Mediators Object System. Amos II is a main memory multi-database system employing a functional data model. Amos II not only has traditional characters of a relational DBMS but also supports object-orientation and distributed databases. As a traditional DBMS, Amos II is made up of a storage manager, a recovery manager, a transaction manager, and a query processor for the query language of Amos II, AmosQL [1]. Amos II can be a stand-alone database server. It can also be set up using its *mediator/wrapper* facilities [2] to access different heterogeneous back-end data sources. In this report the system is used as a stand-alone database server.

2.1 The Amos II Data Model.

An Amos II database is defined in terms of *objects, types*, and *functions*. Next we describe each of these basic system types.

2.1.1 Objects

An Amos II database consists of a set of *objects*. Basically objects are represented in two forms, *surrogate* objects or *literal* objects. A surrogate object has a unique object identifier, *OID*, which is managed by the core system. All user-defined and many system-defined object are surrogate objects. Creation or deletion of surrogate objects is made explicitly through Amos II commands. For example,

- *creating an object:* *create person instances :p;*
- *deleting an object:* *delete :p;*

Unlike surrogate objects, literal objects are self-described system maintained objects that do no have explicit OIDs and are automatically deleted by garbage collection when they are not referenced from any other object. For example, objects representing numbers and strings are literals. *Collections* are also

represented as literal objects. The supported kinds of collections are type *Vector* (ordered sets) and *Bag* (unordered sets with duplicates).

2.1.2 Types

An object is an instance of at least one *type*. The root type is called *Object*. All types inherit from type *Object*. When you create a subtype it will automatically inherit all functions (properties) of its supertype, e.g.

- *create type Person properties(name Charstring, age Integer);*
- *create type Student under Person*
  properties(program Charstring);
- *create type Teacher under Person*
  properties(group Charstring);

Both the types *Student* and *Teacher* inherit the functions *name* and *age* from type *Person*. However, the type *Student* has own function *program* and type *Teacher* has the extra function *group*.

The Amos II type hierarchy is displayed in Figure 2-1.

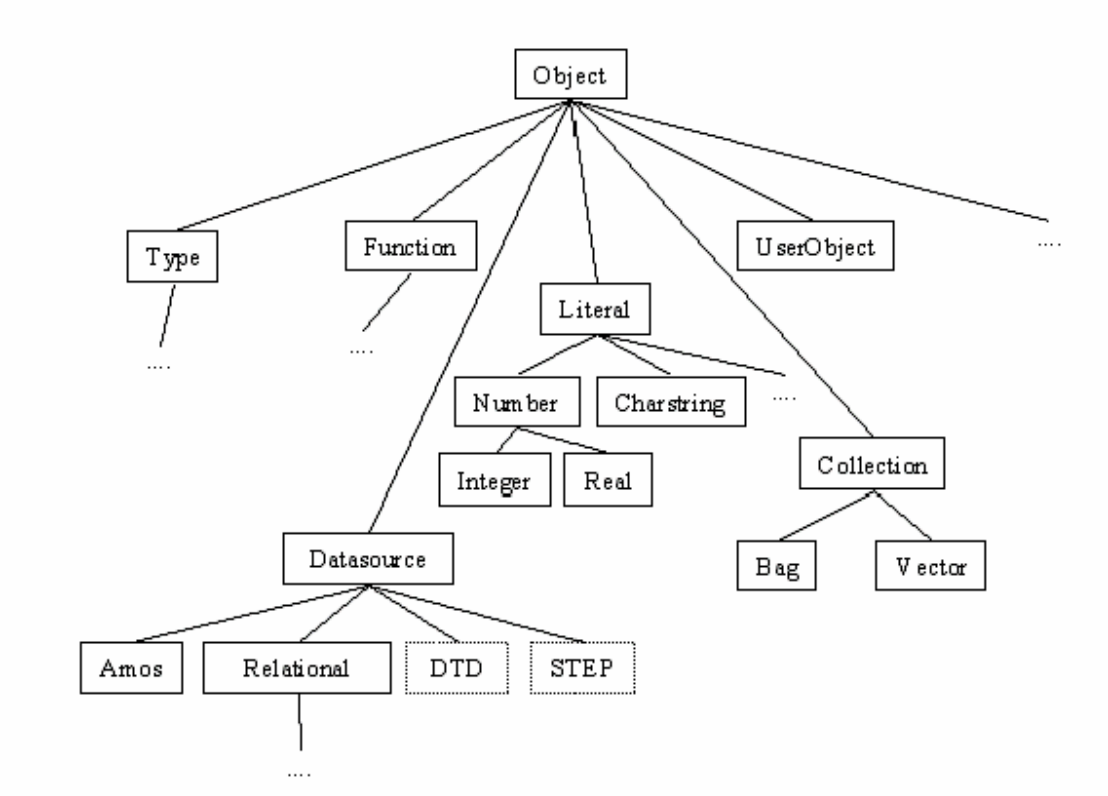

Figure 2-1: Part of the Amos II system type hierarchy

2.1.3 Functions

Functions describe attributes, relationship among different objects, views on objects, and stored procedures for object. All defined functions are instances of type named *Function*. The *signature* of a function represents its name, arguments and results, for example:

- *name(Person p) -> Charstring nm*

The *implementation* of a function describes how to compute results of a function for given parameters. There are five kinds of functions:

1) *Stored functions* which mainly describe objects' properties and whose extents are explicitly stored in the database.

2) *Derived functions* which are defined by a query statement.

3) *Foreign functions* which are defined in some regular programming language. Amos II supports C/C++, Java, and Lisp.

4) *Stored procedures* which are used to specify computations having side effects such as database updates.

5) *Overloaded functions* which are functions having different implementations depending on the types of their arguments. A *resolvent* is a specific implementation of an overloaded function for a given combination of argument types. Each resolvent must have a unique name and there is a convention used in the system for how to construct unique resolvent names. Assume the following two functions:

- *create function info() -> <Charstring name, Integer age>*
  as select name(p),age(p);
- *create function info(Person p)-> <Charstring name, Integer age> as select name(p),age(p) from Person p;*

The capitalization of function names is insignificant and their resolvent names are capitalized. The unique resolvent names of function *info* are:

- *INFO->CHARSTRING.INTEGER*
- *PERSON.INFO->CHARSTRING.INTEGER*

2.2 Mediator and wrapper

Amos II adopts the mediator/wrapper approach to query heterogeneous data sources. A *mediator* is a software module that exploits encoded knowledge about some sets or subsets of data to create information for a higher layer of applications [2]. A mediator does not store any data. It uses *wrappers* to retrieve data from various data source. A wrapper is a software component which queries a data source and translates data from is into the format used in

the mediator. The Figure 2-2 shows the architecture of a mediator/wrapper system.

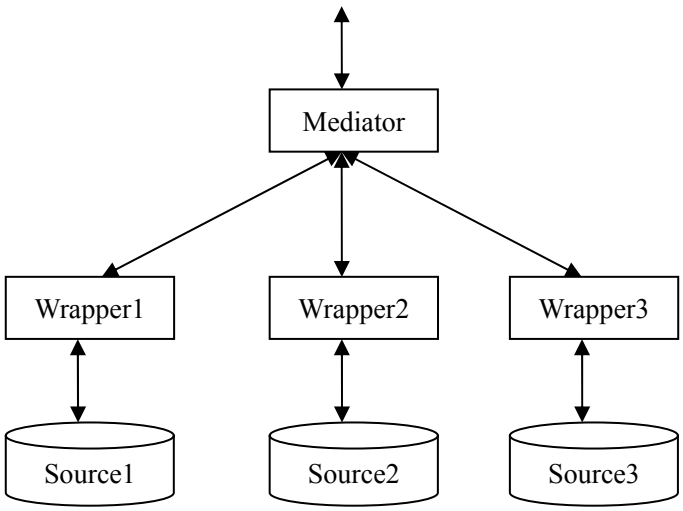

Figure 2-2: The mediator/wrapper architecture

In the mediator/wrapper architecture, a query is first received by a mediator. The mediator will translate the query to sub-queries that the corresponding wrappers can understand. The wrapper uses the sub-queries to retrieve the data and return the result to the mediator. The mediator will collect data returned from different wrappers and process them as the result of the given query.

2.3 External Java interface

Amos II provides external interfaces for the programming languages C/C++, Java, and Lisp. The API description for C/C++ and Lisp is in [3] and for Java is in [4]. We only use the Java API in this project.

The Java API has two kinds of interfaces, called *callin* and *callout* interfaces.

2.3.1 The *callin* Interface

The program developers can use the *callin* interface to make a call or a query to an Amos II database from Java application programs. This interface is similar to other Java database APIs, such as JDBC and ODBC.

With the *callin* interface, there are two different ways to access an Amos II

database.

- The *embedded query* Interface: A query string is passed to the database. Java interface methods are used to convert query results to Java data structures. The embedded query interface is not used in this project.
- The *fast-path* Interface: This interface is used for calling predefined AmosQL functions. It is much faster than the embedded query interface since it avoids dynamically parsing and optimizing the query string at run time.

In this project the fast-path interface is used when the web server invokes exported functions.

2.3.2 The *callout* Interface

Using the *callout* interface, Amos II foreign functions can be implemented as methods written in Java programs. In order to create a Java foreign function, there are two steps:

1. Writing a Java method implementing the function: Such a Java method has two parameter classes, *CallContext* and *Tuple*. The CallContext object is managed by system for error messages and execution control. The *Tuple* object passes arguments and results between Java and Amos II.
2. Defining the signature of the foreign function in AmosQL along with a reference to the Java class implementing it.

In this project the *callout* interface is used for generating WSDL documents, given exported functions. The Amos II server storing the database is instructed to export functions and generate the WSDL file by calling a foreign function.

3. XML and XML Schema

Extensive Markup Language (XML) [6] is a markup language rather than a programming language. It is developed by an XML Working Group formed under the auspices of the World Wide Web Consortium (W3C) in 1996.

3.1 Overview of XML

XML uses a tree-based structure to express data structures. Each node of the structured tree is called *elements* or *entities*. It consists of *makeup* and text. A makeup consists of *start tags* and *end tags*.

An initial example:

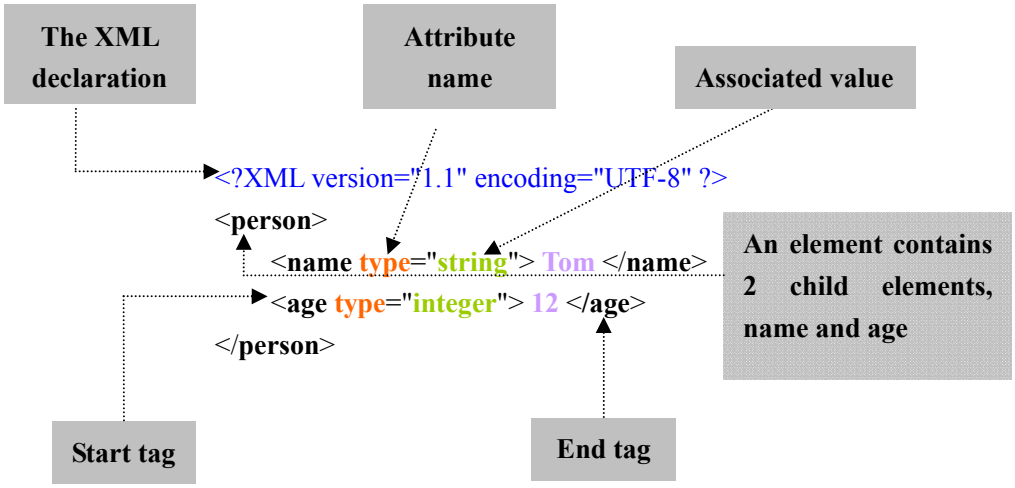

Figure 3-1: Simple example of XML

An XML document has an optional XML *declaration* element. This element specifies what version of XML is to be used and what character encoding is to be used. The nested element immediately following that element contains the data exchanged among systems or applications. Nested elements can have zero or more attributes. Attributes provide meta-information related to the element content. Each attribute consists of a name and an associated value.

XML is designed for storing information by a text-base method in distributed systems. Although XML and Hypertext Markup Language (HTML) are both markup languages and are used to transfer data information via World Wide Web, there are still some differences between them. HTML is concerned about rendering information to browsers not the information itself. Thus, the HTML specification defines a lot of tags, such as <body>, <tr> and <td> to help browsers recognize the web content layout. However, XML cares also about data structures rather than just layouts. It has no predefined tags and users can define their own tags. It is easy to develop software to read and write XML using some of the tools available for this, such as SAX [24] and DOM [25] toolkits.

3.2 Valid and well-formed XML documents.

An XML document is *well-formed* if it complies with some XML's syntax rules [9]:

- One and only one *root element* exists for the XML document. However, the XML declaration, processing instructions, and comment elements can precede the root element.
- Non-empty elements are delimited by both a start-tag and an end-tag.
- Empty elements should be marked with an empty-element (self-closing) tag, such as <Empty />. This is equal to <Empty></Empty>.
- All attribute values are quoted, either single (') or double (") quotes. Single quotes close a single quote and double quotes close a double quote.
- Tags may be nested but must not overlap. Each non-root element must be completely contained in another element.
- The document complies with its character set definition. The charset is usually defined in the XML declaration but it can be provided by the transport protocol, such as HTTP. If no character set is defined, usage of a Unicode encoding is assumed, defined by the Unicode Byte Order Mark. If the mark does not exist, UTF-8 is the default.

All well-formed XML documents that comply with grammar constraints is said to be *valid*. There are several languages to specify the constraint grammar. XML Schema and Document Type Definitions (DTDs) are two often used. This project does not use DTDs XML-Schema provides a syntax to define user defined XML data types. We introduce XML Schema in section 3.4.

3.3 XML Namespaces

Each XML element has its own specified name. Sometime name conflicts will happen when we try to put two or more elements or attributes with the same name into the XML document, while they have different purposes and structure. For this, W3C provides *XML namespaces* [20] to eliminate name collisions among elements or attributes.

An XML namespace is a collection of element type names and attribute names. It is identified by a globally unique identifier reference called a URI. This URI is not as same as the Internet URL although mostly URIs in XML documents are presented as an internet web address.

An element type name or attribute name from a namespace appears in other XML documents as a *qualified name* referred to as a *QName*. In the SOAP or WSDL specifications QNames are widely used. A QName is composed of a *prefix*, a *delimiter*, and a *local* part. The prefix refers to the namespace and the local part is a defined name in that namespace. They are separated by a delimiter, a colon. For instance, in the QName *xsd:string*, 'xsd' is the prefix that refers to the namespace *http://www.w3.org/2000/10/XMLSchema* and '*string*' is a local part in the '*xsd*' namespace.

3.4 XML Schema

With the help of XML Schema it is possible to know how the data structures are represented in an XML document, how data is related to other data, and how to validate the correctness of data. XML Schema is written in XML notation. An XML Schema is also an XML document and follows the rules of XML. By contrast, DTDs are written in non-XML syntax and provide poor data structure definition facilities.

XML Schema is recommended by W3C. Its specification consists of three parts:

XML Schema Part 0: Primer, *http://www.w3.org/TR/XMLschema-0*
XML Schema Part1: Structures, *http://www.w3.org/TR/XMLschema-1*
XML Schema Part2: Datatypes, *http://www.w3.org/TR/XMLschema-2*

Here is an XML Schema example, which defines a *VectorofString* data structure used in this project:

```
<schema xmlns:xsd="http://www.w3.org/2001/XMLSchema">
    <xsd:complexType name="VectorofString">
        <xsd:sequence>
            <xsd:element maxOccurs="unbounded" minOccurs="1" name="member"
type="xsd:string" />
        </xsd:sequence>
    </xsd:complexType>
</schema>
```

Figure 3-2: An XML Schema example

In Figure 3-2, we define a *complexType* element, which can contain a set of elements named *member*. The *type* attribute defines the type of these elements using the QName *xsd:string*. '*string*' is defined as a datatype in *http://www.w3.org/2001/XMLSchema* that is XML Schema definition referred

by the prefix *xsd*. 'xsd' defines some build-in primitive data types, such as *integer*, *double* and *string* conforming to the specification of XML Schema Part 2: Datatypes Second Edition [21].

The structure of XML Schema document

In XML Schema, the first top element is named *schema*. The schema element contains several kinds of child elements [9], for example:

- An *element* declaration element specifies allowable elements in XML instance documents. The declaration element has a name and a type. The type element can be *simpleType* or *complexType*, as described below.
- An *attribute* declaration element specifies allowable attributes of XML instance documents. An attribute element declaration also has a name and a type.
- A *simpleType* declaration element restricts an XML Schema element to have some build-in XML Schema data type (e.g. string or integer). It is constructed by one or more *list*, *union*, or *restriction* elements. Note that these elements are exclusive. A given *simpleType* element can contain only one of those three elements.
- A *complexType* declaration element restricts an XML Schema element to have a *complexType* datatype. In Figure 3-2 *VectorofString* is a *complexType* example. *ComplexType* elements represent complex data structures, such as classes in object-oriented programming languages. It must contain one and only one element named *sequence*, *choice*, *all*, or *group*. It may contain an arbitrary number of attributes or attribute groups.

4. SOAP

Simple Object Access Protocol (SOAP) 1.1 is a recommended message exchange protocol by the World Wide Web Consortium (W3C). You can find the specification of SOAP 1.1 in [7]. We use version SOAP 1.1 in this project.

SOAP 1.1 is a lightweight and simple protocol, which is designed to realize communication between service *requestors* and service *providers* in a decentralized and distributed environment using XML. It does not require a particular application environment, programming language, or transport protocol. Thus, SOAP can potentially use various transport protocols, such as HTTP, SMTP, or FTP. In this project we use the HTTP transport protocol. The

following is an overview of HTTP SOAP message transfer:

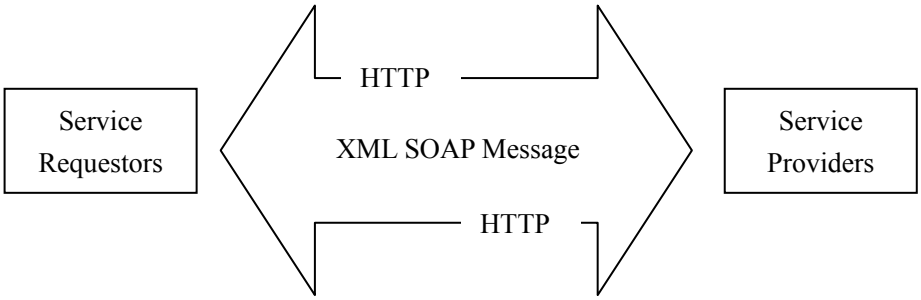

Figure 4-1: Transferring SOAP messages

4.1 The SOAP Message Format

SOAP 1.1 is an XML based protocol. A message can be constructed as shown in Figure 4-2. A SOAP message is an XML document consisting of a top-level element named *envelope*. A SOAP envelope is a container for the SOAP *header* and the SOAP *body*.

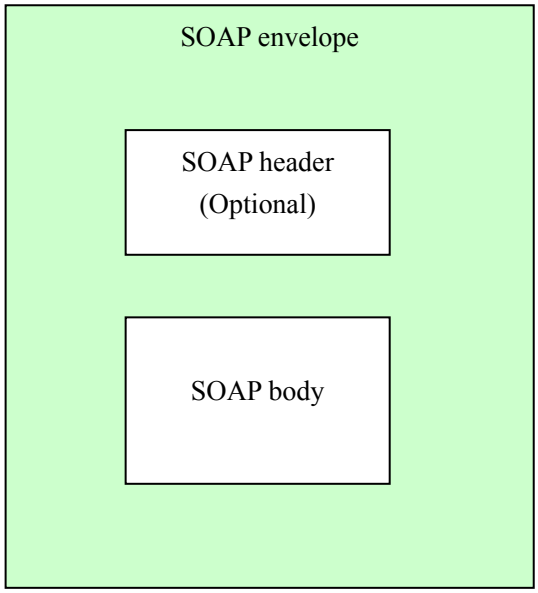

Figure 4-2: A SOAP message without attachments

The SOAP header is an optional section in the SOAP message. It offers a convenient way to provide additional information for a service call. Additional

information can be authentication, transaction information, or other needed information. For example, we can add *password* and *user name* in the header when our service is only available for private customers. We also can put there cookie or session number specifications, or the bill number for each transaction into header. If there is no header, the SOAP body will be the first immediate element in the SOAP envelope.

The SOAP body is a mandatory part of the envelope. It carries the contents of the message. It immediately follows the SOAP header if present. Otherwise, it is the first element in the envelope. All immediate child elements of the body are called *body entries*, and are SOAP message contents. Body entries are independent.

A simple example of a SOAP message:

```
<soapenv:Envelope
    XMLns:soapenv="http://schemas.XMLsoap.org/soap/envelope/"
    XMLns:xsd="http://www.w3.org/2001/XMLSchema"
    XMLns:xsi="http://www.w3.org/2001/XMLSchema-instance">
     <soapenv:Body
          soapenv:encodingStyle="http://schemas.XMLsoap.org/soap/encoding/">
              <tns:INFO XMLns:tns="urn:WSAmos">
                   <P xsi:type="xsd:string">[OID 1048]</P>
              </tns:info>
      </soapenv:Body>
</soapenv:Envelope>
```

Figure 4-3: A SOAP message

4.2 The SOAP encoding

In a distributed and heterogeneous application system, we often have a well-known problem: the data type system of the requestor is not compatible with the type system of the provider. The systems may even be written in different programming languages or run in the different operation systems. The solution of SOAP for this problem is to use *encodings*. The encoding specifies how the message parts are serialized in the message and how to convert the data types.

The SOAP section 5 encoding, defined in the namespace *http://schemas.XMLsoap.org/soap/encoding/,* is a set of data definitions and

encoding styles. It contains all the types found in XML Schema Part 2 datatypes and add many new data types, like *array*. This encoding mechanism is recommended by the SOAP 1.1 specification [7] but is not required; other encoding mechanism can also be defined. In this project we use the SOAP section 5 encoding mechanism as illustrated in Figure 4-3. The element *p* is converted to a string using an attribute *xsi:type*.

4.3 SOAP styles

SOAP support two kinds of styles:

1. Document style:

When using the *document style*, any kind of XML instance can be inserted into the body element in the envelope. The structure of XML instances carried through the SOAP message have no restrictions. The document style is not used in this project.

2. RPC style:

RPC stands for *Remote Procedure Call*. A client can invoke a remote method provided by a service and can get the expected results from the service. The previous example in Figure 4-3 is an RPC style message, which is the only style used in this project. The client calls the function *info* with a parameter named *p*. It expects to get the information of a specified person. The method invocation and the method response are modeled by structs which are added in the SOAP body of the SOAP envelope. Each struct is a compound value in which accessor name is the only distinction among member values, and no accessor has the same name as any other. For instance, the method invocation is viewed as a single struct with the same name as method. The struct contains several accessors which are referenced as parameters to be executed. The structure of a method invocation using RPC style is illustrated in Figure 4-4.

```
<Envelope ….>
    <Header>
        <Header entries>
    </Header>
    <Body ….>
        <MethodName>
            <argument1> Value1 </argument1>
            <argument2> Value2 </argument2>
            ….
        </MethodName>
    </Body>
</Envelope>
```

Figure 4-4: The structure of SOAP message using RPC style

5. WSDL

The Web service description language (WSDL) provides a mechanism to describe the interface of a web service in a structured way. One of the most useful characters is that web service developers can easily and quickly create a corresponding Java program by some software tools. For example, Apache AXIS [17] provides two tools named *Java2wsdl* and *wsdl2Java* to automatically generate Java interfaces to web services described by WSDL documents. Other software vendors, like IBM, SUN, and Microsoft have similar tools.

The Apache AXIS framework provides servlets for Java-based web service communication between clients and servers. *Java2wsdl* generates a WSDL file from a Java interface file and *wsdl2Java* generates Java client-side and server-side binding stubs from a WSDL file. Using the generated WSDL-file and Java code, programmers do not need to care about the communication. They just need to write Java code to call the web service operations and insert them into the generated Java stub code. The intent with AXIS is to make it very simple to split a given Java program into a client part and a server part, which communicate using WSDL/SOAP.

One problem with AXIS is that the server has to be recompiled and redeployed whenever new operations are to be exported. This is a problem in a dynamic system where new interfaces need to be added to a system where the server is always up-and-running. Furthermore, application deployment is somewhat error prone. For example, new database applications might require new server

operations that retrieve particular data from the server. To avoid redeployment one may make a static interface that can handle any supported web service using exactly the same interface. Thus there is no application semantics for the operation specified in the interface. Such a generic interface for Amos II servers was previously developed. It had a general web service operation called *callFunction* that allowed *any* Amos II function to be called with *any* argument list. The client had to dynamically build the argument list as a data structure and provide the function name as a string. Thus the semantics of Amos II functions exported through the web service were not exposed through the WSDL document as every application used exactly the same WSDL document. By contrast, in the present project every exported Amos II function has its own WSDL document that exports the semantics (the signature) of the function.

In a WSDL file some standard type definition namespaces are used. The following namespaces will be used in this report.

Prefix	URI namespace	Definition
wsdl	http://schemas.XMLsoap.org/wsdl/	WSDL namespace for WSDL framework.
soap	http://schemas.XMLsoap.org/wsdl/soap/	WSDL namespace for WSDL SOAP binding.
HTTP	http://schemas.XMLsoap.org/wsdl/http/	WSDL namespace for WSDL HTTP GET & POST binding.
SOAP-ENC	http://schemas.XMLsoap.org/soap/encoding/	Encoding namespace as defined by SOAP 1.1.
SOAP-ENV	http://schemas.XMLsoap.org/soap/envelope/	Envelope namespace as defined by SOAP 1.1.
xsi	http://www.w3.org/2000/10/XMLSchema-instance	Instance namespace as defined by XSD.
xsd	http://www.w3.org/2000/10/XMLSchema	Schema namespace as defined by XSD.
tns		namespaces defined by developers

Table 5-1: List of common name spaces

Figure 5-1 illustrates a WSDL document structure. It has a root element, *definitions*, and six major elements.

- The *definitions* is the root element of the whole WSDL document. It is a container for all other WSDL elements.
- The *types* specifies a container for new XML Schema type definitions

used in the service.

- The *message* is a specification of the data format to be transferred.
- The *portType* specifies a collection of operations used by a service.
- The *binding* is a description of the transfer protocol and data types used by the operations in the *Port Type* part.
- The *operation* specifies an operation to be invoked. The operation gives an abstract operation structure in portType and specifies the transform protocol and style in the binding.
- The *service* is a collection of *ports*. Each port describes the network address of a web service.

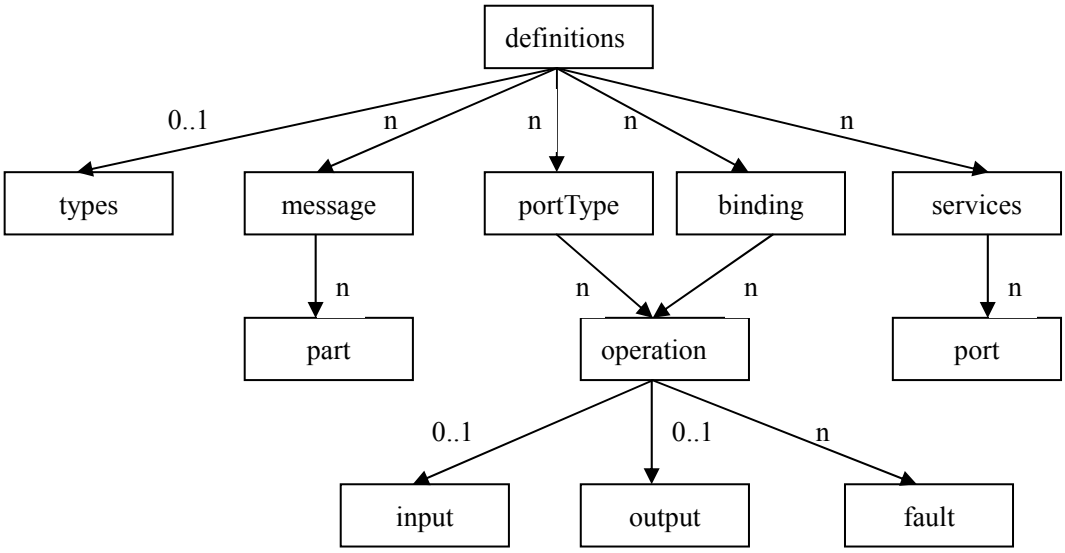

Figure 5-1: WSDL document structure

5.1 The *types* element

The *types* element is a description of the data types used for transferring messages used in the message elements. For cross-platform and cross developing language, WSDL normally uses XML Schema as the type system, but it does not exclude using any other type system. XML Schema allows developers to define arbitrary types. In this project we use XML Schema.

5.2 The *message* element

The *message* element represents arguments and results of an operation. Each *message* element has an attribute *name*, and several *part* elements.

In SOAP RPC style, each *part* element represents a parameter of arguments or results. It has three attributes:

- The *name* attribute specifies the name of the part. It is mandatory and unique within the message.
- The *element* attribute describes the data type of the arguments or results of an operation using the element declaration (section 3.4) in XML Schema.
- The *type* attribute defines the data type of arguments or results of an operation using the *simpleType* or a complex declaration (section 3.4) in XML Schema. The element attribute and the type attribute are exclusive in the part element. It means a part element only has one of these two attributes.

5.3 The *portType* and *operation* elements

In a WSDL file, operation elements are grouped together as child elements of a *portType* element that describes web service operations. A *portType* element is treated as a service interface container in which functions to be invoked are described. An operation element is treated as a method in Java or C/C++.

A *portType* element has one attribute, *name*. The value of this attribute is mandatory and unique within the whole WSDL document.

An operation element has a *name* attribute. In the WSDL1.1 specification, there is no restriction about naming an operation element, but the value of this attribute is conventionally the name of the method implementing the operation. No matter which programming language is used to realize the web service operation, the operation element should include some or all following elements:

- The *input* element:
- The *output* element:
- The *fault* element:

These three elements abstractly specify the arguments, results, and error (exception) of one function respectively. They have two attributes:

- The *name* attribute specifies the name of the operation, which has to be unique within the enclosing *portType*.
- The *message* attribute specifies the message element describing the message of the operation element.

WSDL has four different *operation types* that a web service can support, as listed in Table 5-2. In this project we use the *Request-response* operation only.

Type	Description
One-way Operation	The endpoint service just receives the message. A one-way operation only has an input element. The one-way operation in the endpoint service just receives a request from the service requestor and does not have any response.
Request-response Operation	The endpoint receives a message and sends the results to the client side. A request-response operation has one input element, one output element, and zero or more fault element. The order of these elements is significant.
Solicit-response Operation	The endpoint sends a message and receives the results. A solicit-response operation has one output element, one input element, and zero or more fault element. Again, the order is significant. Compared with the request-response operation, the endpoint service first sends the message rather than receiving a message if it is a solicit-response operation.
Notification Operation	The endpoint just sends the message. There is only an output element. The notification operation in the endpoint service sends a message to the service requestor and does not receive any message.

Table 5-2: Operation types

Only the request-response operation type is used in this project.

In some WSDL files, there may be no *name* attributes in the input or output elements and then default names are used. The default names depend on the operation name. If this operation is a one-way operation or a notification operation, the default name is the name of the operation. If it is a request-response operation or a solicit-response operation, the default is the operation name appending “Request”/”Solicit” or “Response”/”Result”, respectively [8]. The fault element however has no default name and must be given a unique name among all of fault elements.

WSDL 1.1 supports operation overloading. Overloading is a type of polymorphism in which some or all of functions with the same function name are invoked based on the data type of the input parameters. WSDL 1.1 allows overloading operations with the same operation name but these operations must have different input and output names. In other words, the naming of an operation is a combination of the operation name, the input name, and the output name. The combination must be unique within the enclosing *portType* element. In addition, we can also add *parameterOrder* attribute in the operation element although this is optional in WSDL 1.1. The value of the *parameterOrder* attribute is a list of message parts separated by a single space. This list reflects the order in which the parameters appear in the operation signature.

In this project overloaded operations represent overloaded Amos II functions. These functions have the same operation name. Since the input name and output name must be unique, the system generates unique input name and output name by enumerating them.

5.4 The *binding* element

In a WSDL file, the *portType* and the *operation* elements just give an abstract description of a service. They do not tell service users what protocol should be used to transport the SOAP message and what style should be use during transferring SOAP message. The *binding* element does this. The structure of a *binding* element matches the structure of the *portType* element. It has a set of *operation* elements and these operation elements have *input*, *output*, or *fault* elements. A binding element has two mandatory attributes:

- The *name* attribute specifies the name of the binding. It must be unique.
- The *type* attribute is mandatory. The value of the type attribute is a QName that refers to the *portType* element in a namespace.

Since SOAP is the most commonly used communication protocol for hweb services, the WSDL specification describes a set of *extensibility* elements used to specify a *SOAP binding*. The extensibility elements specify the concrete information about the binding and its operations. These extensibility elements are defined in an XML namespace, *http://schemas.xmlsoap.org/wsdl/soap*. We use QName to reference these elements .The WSDL specification also defines other extensibility elements that binds a web service to the HTTP protocol [8].

In this work, the binding for SOAP 1.1 is the only one used. Figure 5-3 is an example of a SOAP binding.

```
...
<wsdl:binding name="WebamosSoapBinding" type="tns:WebamosPortType">
   <soap:binding style="rpc" transport="http://schemas.xmlsoap.org/soap/http"/>
      <wsdl:operation name="ARGUMENTS">
         <soap:operation soapAction=""/>
         <wsdl:input name="ARGUMENTSRequestMsg0">
            <soap:body use="encoded" encodingStyle=
                  "http://schemas.xmlsoap.org/soap/encoding/"
                  namespace="urn:WSAmos"/>
         </wsdl:input>
         <wsdl:output name="ARGUMENTSResponseMsg0">
            <soap:body use="encoded" encodingStyle=
                  "http://schemas.xmlsoap.org/soap/encoding/"
                  namespace="urn:WSAmos"/>
         </wsdl:output>
      </wsdl:operation>
</wsdl:binding>
...
```

Figure 5-3: Example of SOAP binding

In Figure 5-3, three kinds of extensibility elements are used named *soap:operation*, *soap:body,* and *soap:binding*, illustrating the generic rules of extensibility elements for SOAP bindings.

Extensiblity elements	Attributes	Description
soap:binding	style, transport	Specifies the protocol information applied to all operations in the *portType* element being bound. *Style* is *RPC* or *document SOAP style* mentioned in section 4.3. *Transport* specifies what protocol is used to carry the SOAP message. HTTP, SMTP, or FTP could be candidates
soap:operation	style, soapAction	Specifies the protocol information applied to the operation as a whole. *Style* is *RPC* or *document SOAP style* mentioned in section 4.3. It will override the style defined in *soap:binding*. *SoapAction* is placed in the *SOAPAction* HTTP header as the part of an http message.
soap:body	parts,	Provides the details on how the message

	use, encodingStyle, namespace	parts appear in the SOAP body portion of the SOAP message. The attribute *parts* indicates which parts will appear within the SOAP body portion of the message. If the *parts* attribute is omitted, all parts of the message are included. The attribute *use* defines whether the message parts are encoded using some encoding rules or not. The encoding rules are given by the *encodingStyle* attribute. The *namespace* attribute supplies the URI for elements that do not have an explicit namespace.

Table 5-3: SOAP binding extensibility elements

5.5 The *service* element and the *port* element

Up to now, we have described how service customers know how to send and receive data about service calls. However, the address of our service is also needed. The *service* and the *port* elements disclose this information.

The *service* element is a set of port elements that define the web service address for a binding. A WSDL file can contain many service elements. Each one has a distinct value of the *name* attribute.

The *port* element locates web service addresses using an extensibility element, *soap:address*. This extensibility element has an attribute *location*, pointing to the service's URI. A port element has a *name* attribute and a *binding* attribute, both of which are mandatory. The value of the name attribute must be unique among all ports. The binding attribute refers to the relative binding element using QName. For example, when a set of operations within the binding element is bound to a network address the name of that binding element must be equal to the type attribute in the port element.

6. Implementation of WSMOS

Figure 6-1 below illustrates the architecture of WSMOS:

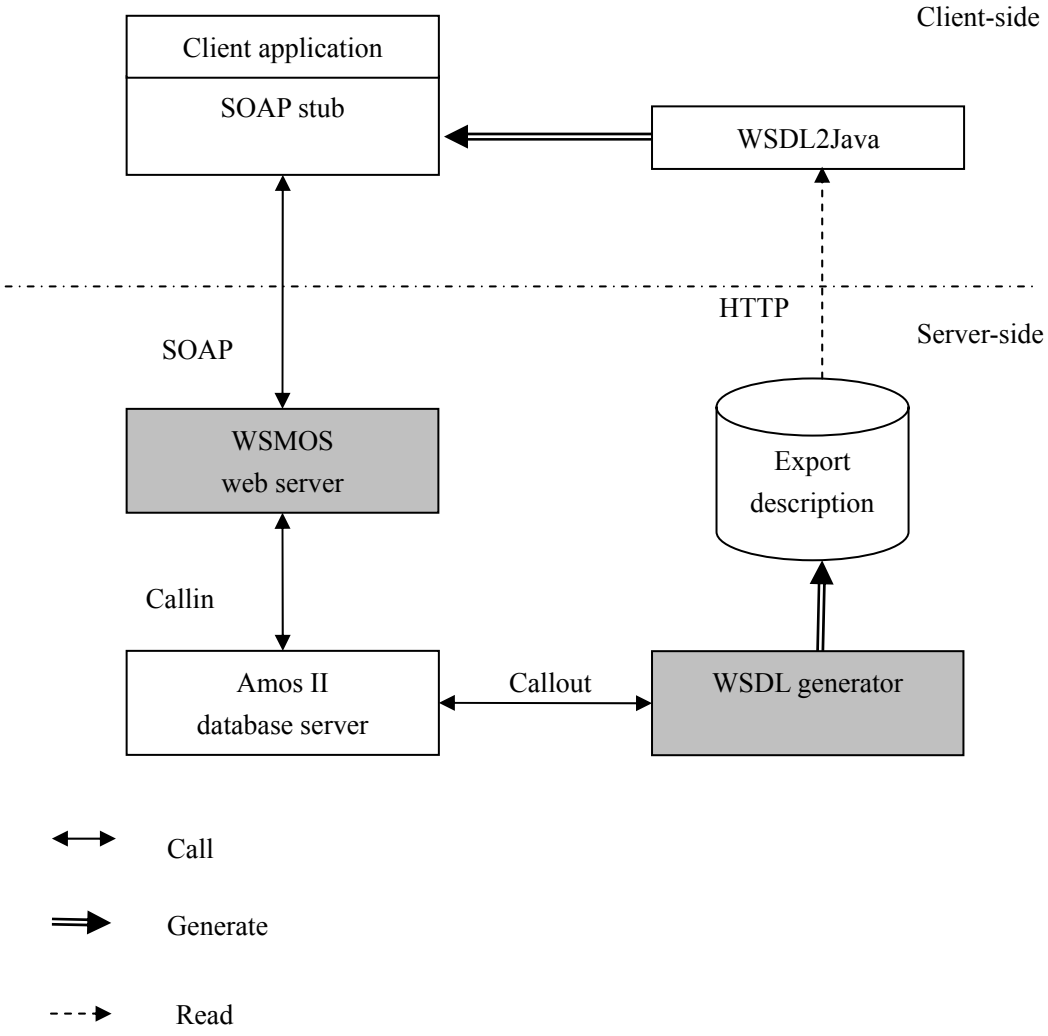

Figure 6-1: The architecture of WSMOS

The WSMOS system includes two parts, the *WSDL generator* and the *WSMOS web server*. They are shown as two dark boxes in Figure 6-1.

The *WSDL generator* is a plug-in (foreign function) to an Amos II server that automatically generates WSDL documents for given exported Amos II functions in the *Amos II database server*. The *export description* is a WSDL document that describes the web service operation interfaces of the exported functions. When a function is exported it is automatically deployed immediately as a web service operation. A *client application* can call any such exported function as an invocation of a web service operation. The programmer simply needs to know the URL of the export description describing the signature of the exported function to call. To construct the communication messages a *SOAP stub* interface is needed. The SOAP stub can be automatically generated from the export description by stub generating tools like *WSDL2Java* [17], which generates web server Java interfaces from a WSDL document. When the

operation of an exported function is called by the SOAP stub, a SOAP message will be transferred to the *WSMOS web server*. The WSMOS web server parses the message and constructs an AmosQL query sent to the *Amos II database server* using the *callin* interface. The database server evaluates the function call and returns the results to the WSMOS web server. The WSMOS web server converts the function call result to a SOAP message sent back as the result of the invocation of the operation. The SOAP stub parses the response SOAP message and extracts the result of the web service operation call.

6.1 The WSDL generator

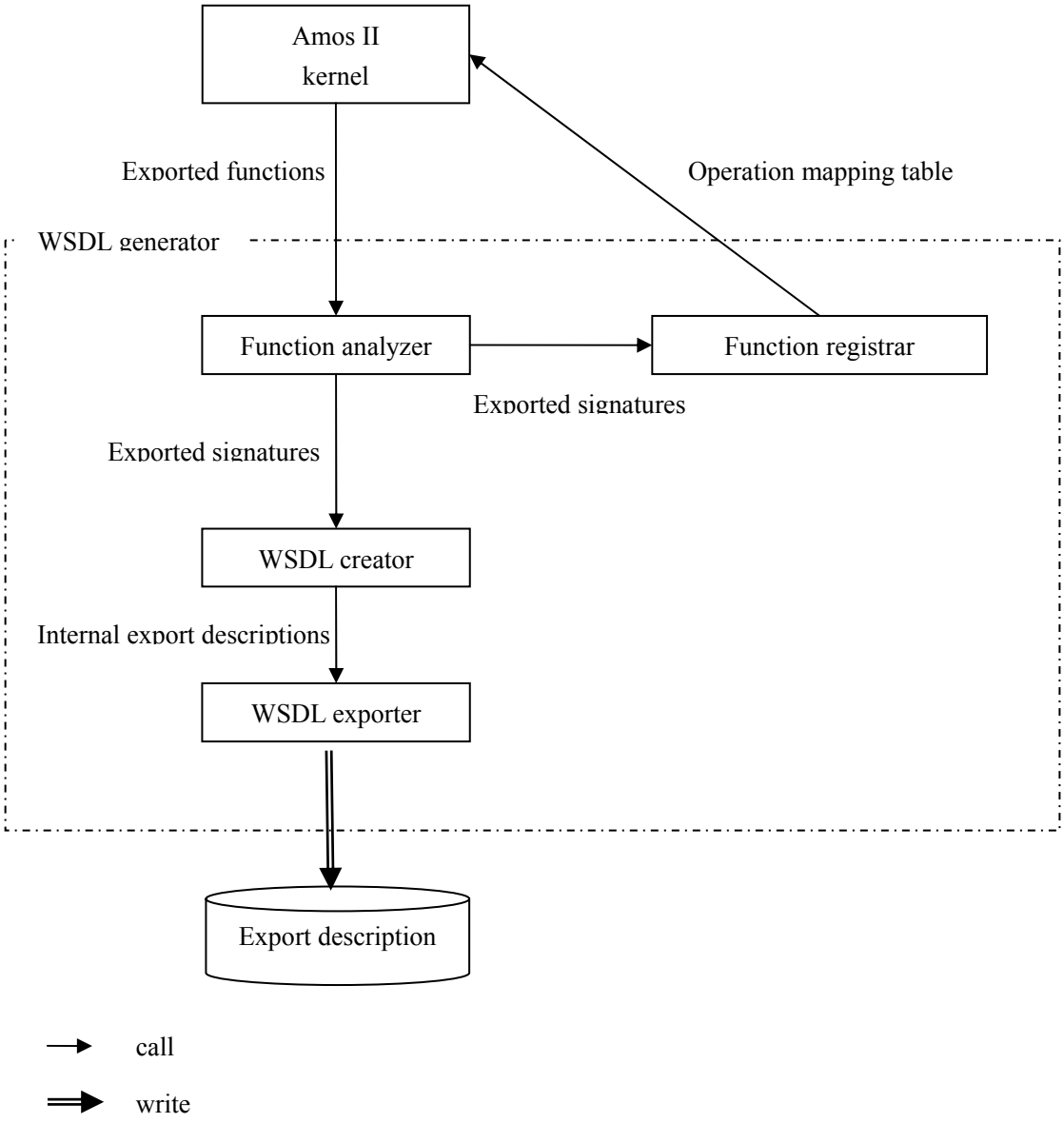

Figure 6-2: WSDL Generator modules

The structure of the WSDL generator is shown in Figure 6-2. Amos II invokes it as an external Java application using the *callout* interface. The WSDL generator consists of four modules, *function analyzer*, *function registrar, WSDL creator*, and *WSDL exporter*.

The *function analyzer* receives a set of exported functions. The function analyzer first checks whether the exported functions are allowed to be exported by looking up a system table. This enables the database administrator to control what functions are exportable, which improves security. It then queries Amos II meta-data for the signature of each function to export and generates *exported signatures* as Java data structures. The signatures consist of the names and types of the functions' arguments and results. They are passed to the *function registrar* and the *WSDL creator.*

The *function registrar* stores the signature of each exported function in its *operation mapping table*. This table is later used when the server constructs a response message for client operation calls. Table 6-1 illustrates the structure of the operation mapping table for a function having the signature *info(Person p)-> <Charstring name, Integer age>*.

funName	argsType	argsName	rettype	Retname
INFO	PERSON	P	CHARSTRING.INTEGER	NAME.AGE
...	...	...	...	...

Table 6-1: Operation mapping table

In the operation mapping table, the combination of *funName* and *argsType* constitutes the compound key. Several system functions are defined to update and query the operation mapping table given *funName* and *argsType*.

The *WSDL creator* dynamically builds the *internal export description* as a DOM data structure in main memory using the WSDL4J Java toolkit [19]. The rules for transforming signatures to WSDL operation descriptions will be discussed in section 6.1.3.

The *WSDL exporter* transforms the DOM representation of the export description into the final *deployed export description*, i.e. the WSDL file describing the exported function interfaces as operations.

6.1.1 SOAP message specification

The WSMOS server is implemented as a web server and we therefore use SOAP over HTTP transport protocol. We always use the SOAP RPC encoding style of message passing since functions are invoked remotely by clients. With this encoding style all data elements are tagged with predefined XML Schema data types. This is required to distinguish between different data types passed as arguments and results from functions. Other styles, such as RPC-literal and document-literal do not provide the required functionality.

6.1.2 Data type mappings

There are two data type mappings needed. First Amos II data types are converted to Java data types using the standard mapping between Amos II and Java [4] as illustrated by Table 6-2. The Java class *Oid* represents any Amos II object.

Amos II data type	⟷	**Java class**
INTEGER		Integer
REAL		Double
CHARSTRING		String
Collection types		Tuple
Others types		Oid

Table 6-2: Data mapping between Amos II and Java

Second the Java data types need to be converted to XSD data types allowing the SOAP RPC encoding style to transfer the messages as XML documents. Table 6-3 illustrates the mapping between the used Java data types and corresponding XSD data types. Java object of class *Tuple* are first converted to *Vector* objects.

Java class	⟷	**XML data type**
Integer		xsd:int
Double		xsd:double
String		xsd:string
Vector		tns:VectorofAnyType or other vector types

Oid		xsd:string with special syntax

Table 6-3: Data mapping between Java and XML

As mentioned in section 3.4, the XML Schema document *http://www.w3.org/2001/XMLSchema* defines many primitive build-in data types provided by W3C. In order to use these predefined primitive data types in that standard namespace, we should use *QName* [20]. For example, *xsd:int*.

If the Java class is *Vector*, we have to map it to our own XML *Vector* type. For this, some new vector types are defined using XML Schema within the *types* element of the WSDL files. The five predefined WSMOS vector types are *VectorofanyType*, *VectorofOID*, *VectorofINTEGER*, *VectorofREAL*, and *VectorofCHARSTRING*. *VectorofanyType* in XML Schema is a generic vector type that can contain any data type. VectorofOID is a set of objects of the Amos II surrogate type Oid. VectorofINTEGER, VectorofREAL and VectorofCHARSTRING respectively represent a set of integer, a set of real, and a set of string. We judge the vector kind based on the definition in Amos II where, e.g., Vector-Integer is a sequence of integers. As the *Vector* type in Amos II represents any ordered set that can contain objects of any Amos II type, it will be mapped to *VectorofanyType*. Figure 6-3 illustrates the data structure of *VectorofanyType*.

```
<definitions>
<types>
<schema >
…
    <xsd:complexType name="VectorofanyType">
        <xsd:sequence>
            <xsd:element maxOccurs="unbounded" minOccurs="1" name="member"
type="xsd:anyType" />
        </xsd:sequence>
    </xsd:complexType>
...
</schema>
</types>
...
</definitions>
```

Figure 6-3: The definition of *VectorofanyType*

The Java class *Oid* is a reference class for any Amos II surrogate data type. In the SOAP message and the WSDL file, we convert the *Oid* object to a special proxy string, beginning with a prefix, "[OID", and ending by a postfix, "]". Between the prefix and the postfix it is the ID-number of the surrogate object. For example, "[OID 1080]" refers to an object stored in the Amos II database whose OID is 1080. In contrast, when the web server receives a string with that particular format, "[OID 1080]", it must convert that string to the corresponding Oid Java class referring to the "#[OID 1080]" object in Amos II. After the web server parses the string and gets the ID-number (1080), it calls Amos II to obtain the corresponding *Oid* Java object representing the Amos II object having the given ID.

6.1.3 Describing an Amos II function as a WSDL operation:

Assuming that we have a function:

create function info(Person p)-> <Charstring name, Integer age>

as select name(p),age(p);

Its signature is *info(Person p)-> <Charstring name, Integer age>*. The semantics of this function is to retrieve a specified person's name and age.

The corresponding WSDL code to describe the signature of the function *info* looks like this:

```
<wsdl:definitions ...>
<wsdl:types>
   <xsd:schema>
        ...
       <xsd:complexType name="INFOReturn0"><xsd:sequence>
          <xsd:element maxOccurs="unbounded" minOccurs="0" name="row">
              <xsd:complexType><xsd:sequence>
                    <xsd:element name="NAME" type="xsd:string"/>
                    <xsd:element name="AGE" type="xsd:int"/>
              </xsd:sequence></xsd:complexType>
          </xsd:element>
        </xsd:sequence></xsd:complexType>
       ...
   </xsd:schema>
</wsdl:types>

 <wsdl:message name="INFOResponseMsg0">
      <wsdl:part name="results" type="tns:INFOReturn0"/>
 </wsdl:message>
 <wsdl:message name="INFORequestMsg0">
      <wsdl:part name="P" type="xsd:string"/>
 </wsdl:message>

 <wsdl:portType name="WebamosPortType">
      <wsdl:operation name="INFO" parameterOrder="P">
         <wsdl:input name="INFORequestMsg0" message="tns:INFORequestMsg0"/>
         <wsdl:output name="INFOResponseMsg0"
                  message="tns:INFOResponseMsg0"/>
      </wsdl:operation>
 </wsdl:portType>
...
</wsdl:definitions>
```

Figure 6-4: WSDL codes for signature of function *info*

In Figure 6-4 we define an operation called *INFO* within the *portType* element.

The *INFO* operation contains an input element and an output element. The input element has an attribute *message* referred to an input message named *INFORequestMsg0*. Figure 6-5 illustrates the structure of the message.

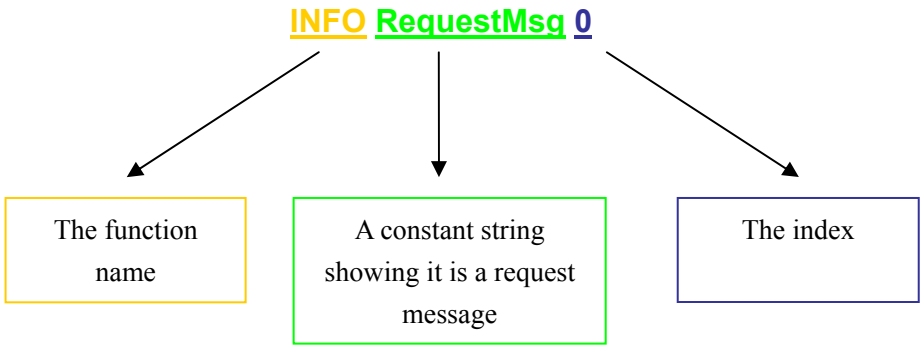

Figure 6-5: The structure of the input element

The *index* is used to support overloading. According to the specification of WSDL 1.1, the combination of the operation name, the input name, and the output name must be unique within the *portType* element. To achieve this, if there are any other operations with the same function name, the WSDL generator will increase the index by one for each new overloaded function resolvent. For example, if there had been another *info* resolvent we would have increased the value of the index to 1, *INFORequestMsg1*.

The output element also has an attribute *message* linked to an output message and its name is *INFOResponseMsg0*. The constant string *RepsonseMsg* corresponds to *RequestMsg* to indicate that this is a response message.

The *INFORequestMsg0* message contains one *part* element since the *info* function has only one argument named *p* and its type is *Person*. Since *Person* is a user defined type, it is represented as an OID proxy string (see section 6.1.2 Data Type Mappings). Therefore, we set its *name* attribute to *P* and its type attribute to *xsd:string* in the *part* element.

The output message contains one *part* element that is linked to a new data type *INFOReturn0* defined in the schema element.

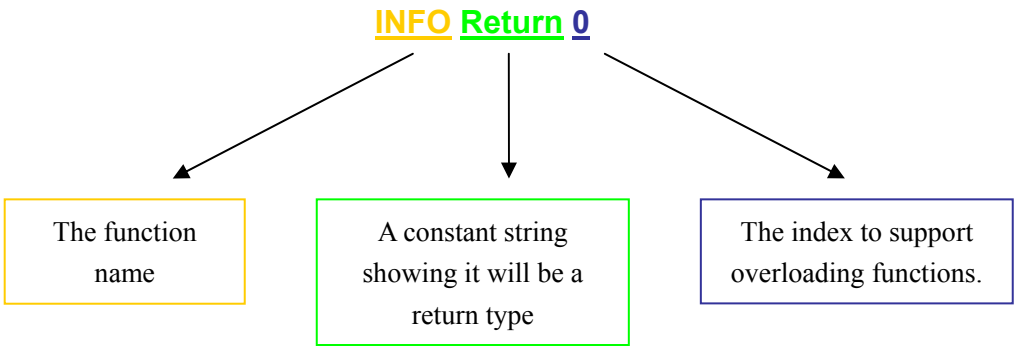

Figure 6-6: The structure of the input element's name

The *INFOReturn0* data structure has a sequence element named *row*. Each *row* element represents a function call result. Since the result of the *info* function is a 2-way tuple, <*Charstring name, Integer age*>, the row element contains two child elements. The first one is named *NAME* and its type is mapped to *xsd:string* since the type of the first result parameter is *Charstring*. The other is named *AGE* and its type is mapped to *xsd:int*.

In general, the WSDL creator has the following conventions in order to correctly describe an (overloaded) Amos II function as an operation in the WSDL document:

1. The *name* attribute of the operation is the generic function name (capitalized). The *parameterOrder* attribute is a list of function argument names (capitalized) separated by spaces.

2. The names of the input element and the input message consist of the function name, the constant string *RequestMsg*, and an index. The index distinguishes overloading functions in the WSDL document. It starts from 0 and is increased by adding one for each new resolvent.

3. The input message contains several *part* elements which represent function arguments. All *part* elements keep the same order as the function arguments. The value of the *name* attribute in each *part* element is the name of the corresponding function argument. The *type* attribute uses the data type mapping between Java and XML in section 6.1.2.

4. The names of the output element and the output message consist of the function name, the constant string *ResponseMsg*, and an index. This index corresponds to the index that appears in the input element and the input message.

5. The output message contains one part element named *results*. Since the result of the function call normally is a collection, the output message structure is more complex than the input message. We can not directly set the result parameters as *part* elements as for input messages. Instead, we define the corresponding result structure in the *types* element in the WSDL document. The part element within the output message using the *type* attribute refers to the complex XML Schema data structure defined in the *types* element in the WSDL document. The value of this *type* attribute is the function name plus the string *Return* and the index of overloaded functions.

6. The *complexType* collection data type holds one query result tuple. It has one attribute, *name*. Each result tuple is represented as an element named *row*. The *row* element also is a *complexType* element. All result parameters are listed under it and keep the same order as the function result tuple specification. For instance, in Figure 6-4 the results tuple of function *info* is <*Charstring name, Integer age*>. Two child elements representing *name* and *age* are added within the row element. Figure 6-7 illustrates the generic structure of results.

```
...
<xsd:complexType name="FunctionName + 'Return'+Index">
  <xsd:sequence>
    <xsd:element maxOccurs="unbounded" minOccurs="0" name="row">
      <xsd:complexType>
        <xsd:sequence>
            <xsd:element name="Result parameter 1" type="xsd:datatype"/>
            <xsd:element name="Result parameter 2" type="xsd:datatype "/>
            ...
        </xsd:sequence>
      </xsd:complexType>
    </xsd:element>
  </xsd:sequence>
</xsd:complexType>
...
```

Figure 6-7: WSMOS result structure

7. The child elements of a *row* element have the two attributes, *name* and *type*, and a text node containing the result value. The value of the *name* attribute is equal to the name of the function result parameter. The value of *type* refers to the data type mapping in section 6.1.2.

6.2 The WSMOS web server

As we are using the HTTP protocol to carry SOAP messages, the WSMOS web server is a Java HTTP SOAP server. Its purpose is to immediately deploy specified Amos II functions as web service's operations when the database administrator exported the specified functions, without restarting the web server or deploying any Java code.

Figure 6-8 illustrates WSMOS web server components.

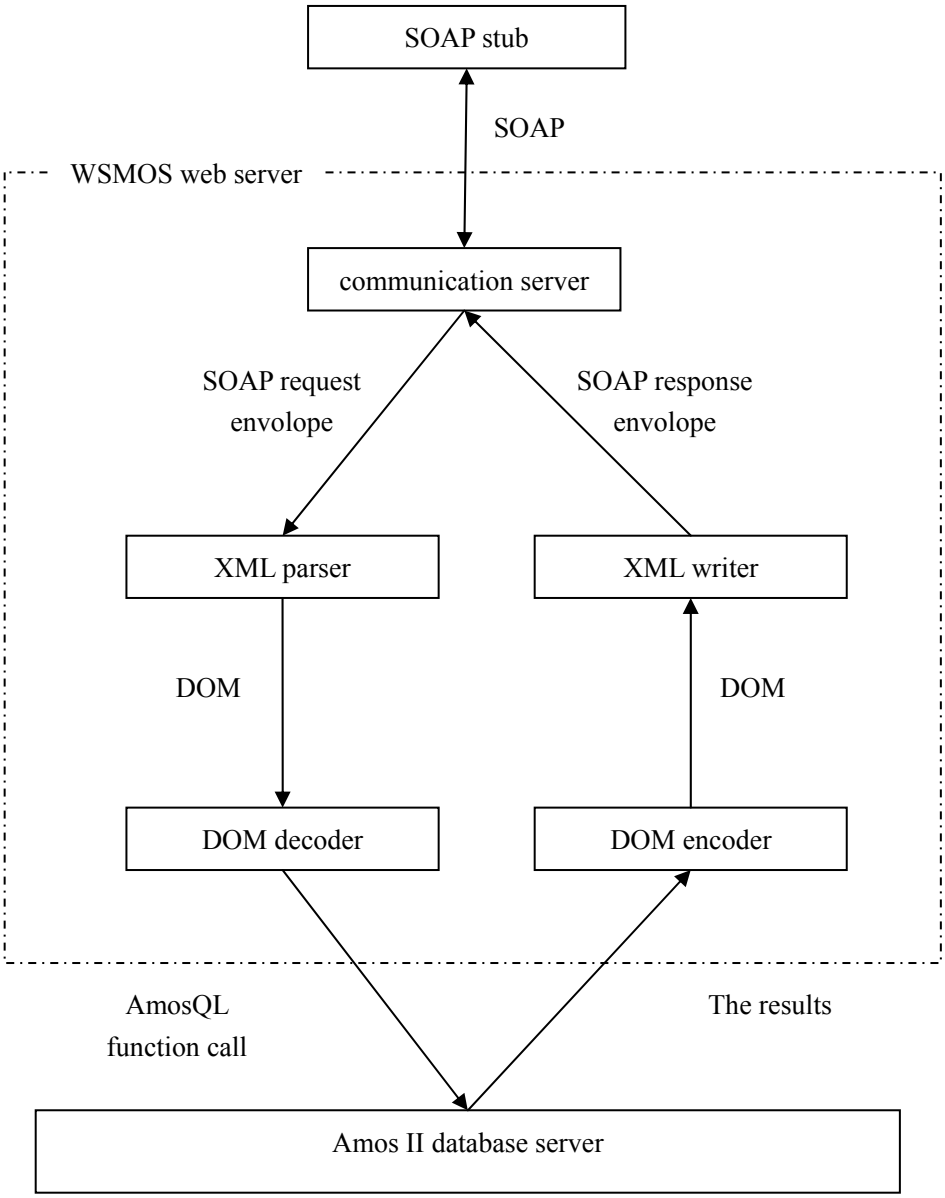

Figure 6-8: The WSMOS web server components

The WSMOS web server consists of a *communication server*, an *XML parser* and *writer*, a *DOM decoder*, and *an encoder*.

The *communication server* first receives a remote call from the *client interface*. The remote call is a RPC SOAP call via the HTTP protocol. The communication server extracts the message content and passes it to the *XML parser*. The XML parser uses the input SOAP envelope to generate a DOM data structure. A *DOM decoder* using the data type mapping rules between XML and Java in section 6.1.2 decodes the DOM data structure, and extracts the function name, along with names, types and values of the function call arguments. When decoding, we use the type information provided by the SOAP message. Thus we do not use the operation mapping table here. After getting results from Amos II, *the XML encoder* uses the operation mapping table and data type mapping rules between XML and Java to encode the result to another result DOM structure. The *XML writer* prints the result DOM structure to the communication server as a SOAP response message and the communication server sends back the SOAP message to the client interface over HTTP protocol.

6.2.1 The communication server

The WSMOS web server is an HTTP SOAP server. Two version of the WSMOS communication server are developed, either a modified *JSoapServer* [16] or the Apache *Tomcat* web server [11].

JSoapServer [16] is a lightweight standalone SOAP web server using two libraries, *Apache Axis* [17] and *QuickServer* [18]. Axis contains an XML parser/writer, a DOM decoder, and a DOM encoder. The *QuickServer* is a library for handling the HTTP protocol. To construct a simple communication server we modified the code of *JSoapServer* by removing the Apache Axis library and just use the *QuickServer* library. We call our communication server *AmosSoapServer*.

Tomcat [11] is another free standalone and cross-platform web server that supports servlets and JSP. A *servlet* is an object that receives a request (*ServletRequest*) and generates a response (*ServletResponse*). Sun Microsystems provides an API package *Javax.servlet.http* to define HTTP subclasses of generic Java servlet classes for requests (*HttpServletRequest*), responses (*HttpServletResponse*), and sessions (*HttpSession*). When using WSMOS with Tomcat we use the servlet mechanism to handle the HTTP SOAP request and response.

Comparing *Tomcat* and *AmosSoapServer*, the latter is a lightweight system

which is simple and uses less memory and CPU time. One *AmosSoapServer* only supports one web server while Apache Tomcat can deploy several web servers simultaneously. Only one web server is needed in this project. A problem with Tomcat is that the whole system will crash when one of web servers crashes. Therefore, we recommend using *AmosSoapServer* as the communication server for WSMOS.

6.2.2 The XML parser

The communication server transfers a string representing a SOAP request envelope to the XML parser for building a DOM data strucure. Conversely, the XML writer will convert a DOM data structure to a string as an output of operation when the DOM encoder returns the results.

In order to analyze, create, and modify SOAP messages, we use a public Java library called *SAAJ* [13], which stands for *SOAP with Attachments API for Java*. SAAJ enables programmers to quickly produce and consume SOAP messages conforming to the SOAP 1.1 specification. It is developed by the JSR 67 export group [23].

6.2.3 The DOM decoder

In order to invoke the correct function, we must analyze a received SOAP message. The XML parser converts the SOAP message to DOM. The following example is a SOAP message envelope using RPC/Encoding style:

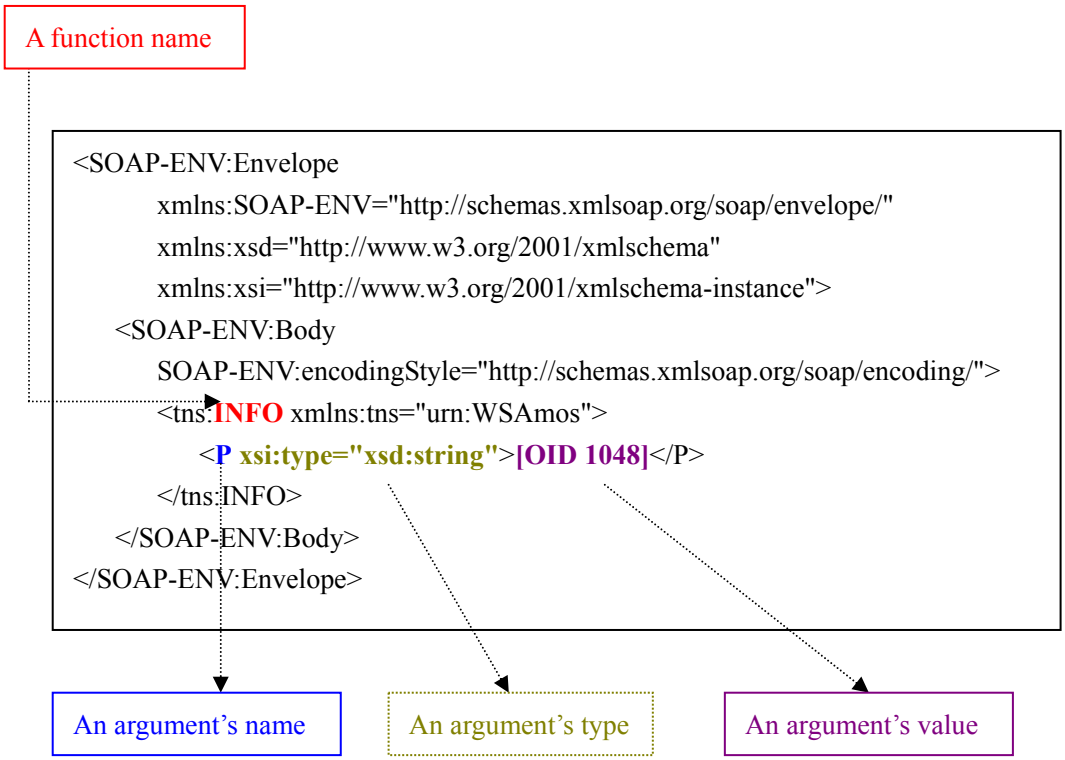

Figure 6-9: A WSMOS SOAP message

In the SOAP message example in Figure 6-9, the first immediate child element of the body is *INFO*. It has a single child element named *P*. The text node of *P* is *[OID 1048]*. Therefore, we know that the function is named *INFO* and the argument name is *P*. The *xsi:type* attribute of element *P* is *xsd:string* and its text node is a proxy string that starts with a "*[OID*" and ends with "*]*". A new Java *Oid* object is created, which is a surrogate for the Amos II object "*#[OID 1048]*" using the Amos II Java interface [4].

Figure 6-10 shows the generic structure of a SOAP request message using RPC/encoding:

```
< Envelope ...>
    <Header...> ... </Header>
    < Body >
        <FunctionName>
            <Argument1 xsi:type="xsd:datatype"...> value1 </Argument1>
            <Argument2 xsi:type="xsd:datatype"...> value2 </Argument2>
            <Argument3>

                <element1 xsi:type="xsd:datatype" ...> value < /element1 >
                <element2 xsi:type="xsd:datatype "...> value < /element2 >
                ...
            </Argument3>
            .....
        </FunctionName>
    </Body>
<Envelope>
```

Figure 6-10: Generic structure of WSMOS SOAP request message

The body element only contains one child element which is named by the name of the function to be invoked. That element consists of zero or more child elements that represent the input parameters.

Elements using primitive data types have an attribute "*xsi:type*" and one text node. They have no child element. The value of "*xsi:type*" is an XML data type. We use the data type mapping mechanism in section 6.1.2 to decode XML data types as Java data types.

In difference to the primitive data types, the parent element of a collection data type does not have any "*xsi:type*" attribute or text node. It contains a number of child elements. These child elements can be primitive data types or collection data types. If the child element is a primitive data type, we decode them by the rules mentioned in the previous paragraph. If it is a collection data type, we will recursively parse it.

6.2.4 The DOM encoder

Function results are sent back to the DOM encoder as a Java *Tuple* object through the Amos II Java interface. The DOM encoder will convert it to a Java *Vector* type. The SOAP response message in Figure 6-11 is based on the result of *info* function call. The function call result is <*Milena, 30*>.

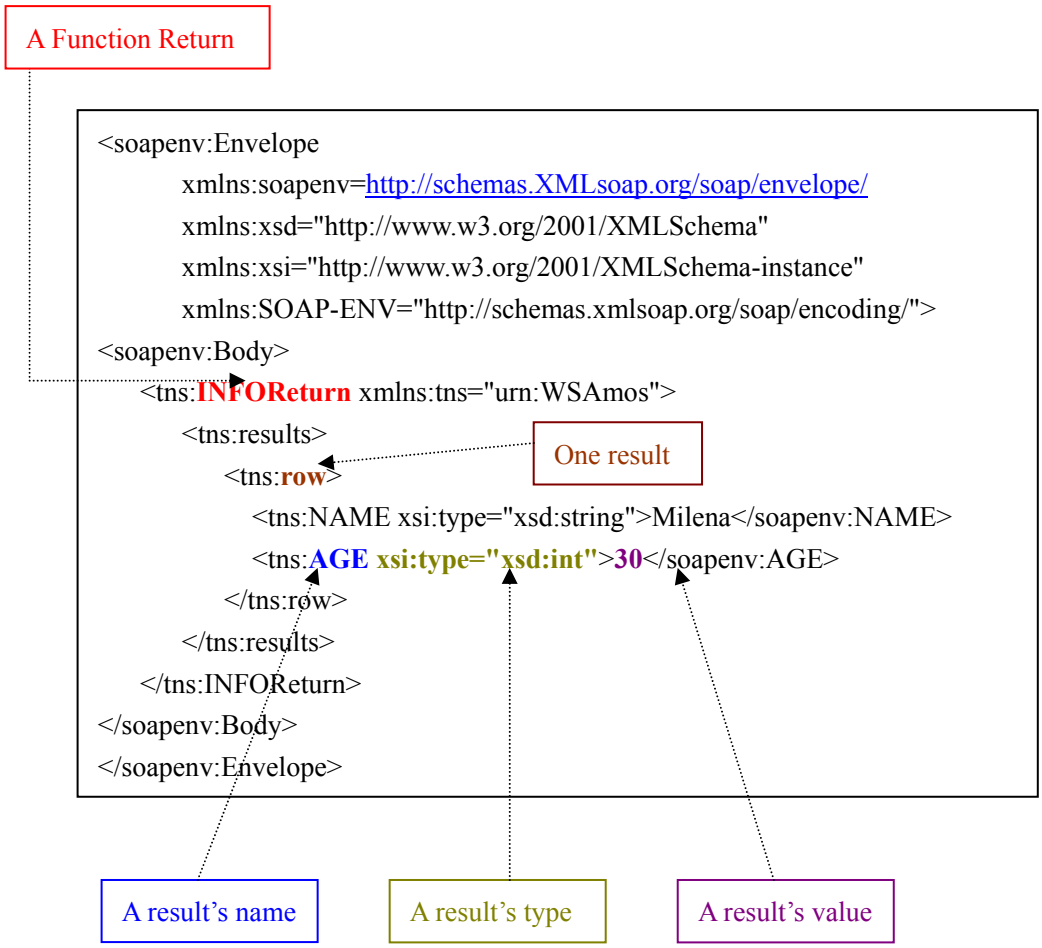

Figure 6-11: A SOAP response message

Using the convention of SOAP RPC communication style, the DOM encoder will first insert an *INFOReturn* element in the body element as its immediate child element. Under that element, the DOM encoder will build the results using the structure described in the WSDL file. The element results and its child row are added. The row element refers to a result tuple. A tuple of result parameters will be stored in a row element. In order to retrieve the signature of the function call, the DOM encoder queries the operation mapping table in the Amos II database server. It contains all necessary meta-data about the function call signature. The signature of *info* function is *info(Person p)-> <Charstring name, Integer age>*. Using the operation mapping table, the DOM encoder thus knows that the name of the first result parameter is *NAME* and its type is *Charstring*, while the second parameter name is *AGE* and type is *Integer*. Consequently, the DOM encoder creates two corresponding child elements within the row element.

The generic structure of a SOAP response message is shown in Figure 6-12.

```
< Envelope>
< Body>
    < FunctionName+"Return" >
        <results>
            <row>
                < parameter1 of result1 > value1 </ parameter1>
                < parameter2 of result1> value2 </ parameter2>
                 ...
            </ row>
            <row>
                < parameter1 of result2> value1 </ parameter1>
                < parameter2 of result2> value2 </ parameter2>
                ...
            </row>
            ...
        </ results>
    </ FunctionName + "Return">
</Body>
</Envelope>
```

Figure 6-12: Generic structure of WSMOS SOAP response message

The following rules are applied when the DOM encoder builds a SOAP response message:

1. The immediate child element in the body element is a function name plus a postfix string "*Return*".

2. The function call result is a sequence of elements named *results*. Thus, the immediate child element of the body has only a child element named *results*.

3. The *results* element contains zero or more elements named *row*. If the query is an empty scan, there is no row element.

4. Each child element of the row element represents one function result tuple element. The names and types of the function results are stored in the operation mapping table. The DOM encoder queries these meta-data and uses them to build all child elements of the row element. The encoder will map the

Java data type of each result parameter to an XML data type. It follows the mappings in section 6.1.2.

6.2.5 Handling exception:

In most programming languages there is a mechanism to throw exceptions. For example, Java has the *try* and *catch* statements to catch exceptions. SOAP has a similar mechanism. It uses the *fault* element to send back an error or exception code via a SOAP message to the SOAP stub when an exception occurs. The *fault* element is contained in the body element and a body element may have only one fault element. When the client interface receives the error message, the clients will know where the problem is and can fix the problem.

Figure 6-13 illustrates a SOAP error message.

```
<soapenv:Envelope
        xmlns:soapenv="http://schemas.xmlsoap.org/soap/envelope/"
        xmlns:xsd="http://www.w3.org/2001/xmlschema"
        xmlns:xsi="http://www.w3.org/2001/xmlschema-instance"
        xmlns:SOAP-ENV="http://schemas.xmlsoap.org/soap/encoding/">
<soapenv:Body>
  <soapenv:Fault>
    <faultcode xmlns:SOAP-ENV="http://schemas.xmlsoap.org/soap/envelope/">
        SOAP-ENV:Server
    </faultcode>
    <faultstring>callin.AmosException</faultstring>
    <faultactor></faultactor>
    <detail>
      <tns:FaultDetail xmlns:tns="urn:WSAmos">
          ERROR BAD_REQUEST. Function, info, is not exported.
      </tns:FaultDetail>
    </detail>
  </soapenv:Fault>
</soapenv:Body>
</soapenv:Envelope>
```

Figure 6-13: A SOAP error message

If, when the client invokes a function, the information of *info* function does not

exist in the operation mapping table the server will throw an Amos exception and build a *fault* element. In the *FaultDetail* element, the error message is ”*ERROR BAD_REQUEST. Function INFO is not exported*“. The client system can read the error message to know that the *info* function is not exported. The error message is constructed following the specification of WSDL 1.1 [8].

7. Summary and future works

The WSMOS project implements a mechanism of automatic deployment of web services for data access. The *WSDL generator* creates a web service interface description (WSDL document) for specified exported Amos II functions conforming to the WSDL 1.1 specification. The WSDL document defines the signature of exported functions, the service URI, and how to transfer SOAP messages. It does not limit the client to use a particular platform or programming language. For example, a C++ client can invoke the operation of the WSMOS web server written in Java via the SOAP message, or the client can be running on a different operating system. On the server side, the WSMOS web server receives incoming SOAP messages sent by an operation call, dynamically interprets the received SOAP message, and queries Amos II through the *callin* interface. From the result of the function call the system builds a SOAP response message and sends it back to the client.

Web services include not only WSDL and web server applications but also a service to publish WSDL files. The Universal Description, Discovery, and Integration (*UDDI*) registry provides a place to store different WSDL files from different web service providers. Web clients can easily find their web service through the UDDI. UDDI registries are provided by IBM, Microsoft, Sun, and other technology companies. They contain a programmatically accessible description of services and define the programming model and schema. Future work could include developing an automatic application to render the WSDL file to some UDDI registries.

Other future work would deal with security since every system have sensitive resources that can be accessed by many users, or resources that traverse unprotected and open networks, such as the Internet. We can add security at the transport layer, the message layer, and the application layer.

Appendix A: A WSDL document with overloading operations

Assume the following two resolvents of the overloaded Amos II function *info* with signatures (Amos II function names in small letters, while types has first letter capitalized):

info() -> <Charstring name,Iinteger age>.

info(Person p) -><Charstring name, Integer age>

The corresponding generated WSDL document is:

```
<?xml version="1.0" encoding="UTF-8" ?>
<wsdl:definitions name="Webamos" targetNamespace="urn:WSAmos"
          xmlns:tns="urn:WSAmos"
          xmlns:wsdlsoap="http://schemas.xmlsoap.org/wsdl/soap/"
          xmlns:xsd="http://www.w3.org/2001/XMLSchema"
          xmlns:SOAP-ENV="http://schemas.xmlsoap.org/soap/encoding/"
          xmlns:wsdl="http://schemas.xmlsoap.org/wsdl/">

<wsdl:types>
<xsd:schema targetNamespace="urn:WSAmos"
         xmlns="http://www.w3.org/2001/XMLSchema">
 <xsd:complexType name="VectorofanyType">
 <xsd:sequence>
      <xsd:element maxOccurs="unbounded" minOccurs="1" name="member"
           type="xsd:anyType" />
  </xsd:sequence>
  </xsd:complexType>
 <xsd:complexType name="VectorofOID">
 <xsd:sequence>
      <xsd:element maxOccurs="unbounded" minOccurs="1" name="member"
           type="xsd:string" />
  </xsd:sequence>
  </xsd:complexType>
   <xsd:complexType name="VectorofINTEGER">
   <xsd:sequence>
      <xsd:element maxOccurs="unbounded" minOccurs="1" name="member" type="xsd:int"
           />
  </xsd:sequence>
  </xsd:complexType>
 <xsd:complexType name="VectorofREAL">
 <xsd:sequence>
```

```
      <xsd:element maxOccurs="unbounded" minOccurs="1" name="member"
          type="xsd:double" />
  </xsd:sequence>
  </xsd:complexType>
 <xsd:complexType name="VectorofCHARSTRING">
 <xsd:sequence>
      <xsd:element maxOccurs="unbounded" minOccurs="1" name="member"
          type="xsd:string" />
  </xsd:sequence>
  </xsd:complexType>
 <xsd:complexType name="INFOReturn0">
 <xsd:sequence>
     <xsd:element maxOccurs="unbounded" minOccurs="0" name="row">
     <xsd:complexType>
     <xsd:sequence>
        <xsd:element name="NAME" type="xsd:string" />
        <xsd:element name="AGE" type="xsd:int" />
    </xsd:sequence>
    </xsd:complexType>
    </xsd:element>
  </xsd:sequence>
  </xsd:complexType>
 <xsd:complexType name="INFOReturn1">
 <xsd:sequence>
     <xsd:element maxOccurs="unbounded" minOccurs="0" name="row">
     <xsd:complexType>
     <xsd:sequence>
        <xsd:element name="NAME" type="xsd:string" />
        <xsd:element name="AGE" type="xsd:int" />
     </xsd:sequence>
     </xsd:complexType>
     </xsd:element>
  </xsd:sequence>
  </xsd:complexType>
</xsd:schema>
</wsdl:types>

<wsdl:message name="INFOResponseMsg0">
  <wsdl:part name="results" type="tns:INFOReturn0" />
</wsdl:message>
<wsdl:message name="INFORequestMsg0">
  <wsdl:part name="P" type="xsd:string" />
</wsdl:message>
<wsdl:message name="INFOResponseMsg1">
```

```
  <wsdl:part name="results" type="tns:INFOReturn1" />
</wsdl:message>
<wsdl:message name="INFORequestMsg1" />

<wsdl:portType name="WebamosPortType">
   <wsdl:operation name="INFO" parameterOrder="P">
      <wsdl:input name="INFORequestMsg0" message="tns:INFORequestMsg0" />
      <wsdl:output name="INFOResponseMsg0" message="tns:INFOResponseMsg0" />
  </wsdl:operation>
   <wsdl:operation name="INFO" parameterOrder="">
      <wsdl:input name="INFORequestMsg1" message="tns:INFORequestMsg1" />
      <wsdl:output name="INFOResponseMsg1" message="tns:INFOResponseMsg1" />
  </wsdl:operation>
</wsdl:portType>

<wsdl:binding name="WebamosSoapBinding" type="tns:WebamosPortType">
  <wsdlsoap:binding style="rpc" transport="http://schemas.xmlsoap.org/soap/http" />
   <wsdl:operation name="INFO">
      <wsdlsoap:operation soapAction="" />
      <wsdl:input name="INFORequestMsg0">
          <wsdlsoap:body use="encoded"
                encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"
                namespace="urn:WSAmos" />
      </wsdl:input>
      <wsdl:output name="INFOResponseMsg0">
          <wsdlsoap:body use="encoded"
                encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"
                namespace="urn:WSAmos" />
      </wsdl:output>
    </wsdl:operation>
   <wsdl:operation name="INFO">
      <wsdlsoap:operation soapAction="" />
      <wsdl:input name="INFORequestMsg1">
         <wsdlsoap:body use="encoded"
                encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"
                namespace="urn:WSAmos" />
      </wsdl:input>
      <wsdl:output name="INFOResponseMsg1">
         <wsdlsoap:body use="encoded"
                encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"
                namespace="urn:WSAmos" />
      </wsdl:output>
    </wsdl:operation>
</wsdl:binding>
```

```
<wsdl:service name="WebamosService">
    <wsdl:port name="WebamosPort" binding="tns:WebamosSoapBinding">
      <wsdlsoap:address location="http://130.238.11.189:8082/wsmos/service/AmosServlet" />
    </wsdl:port>
</wsdl:service>
</wsdl:definitions>
```

Appendix B: A SOAP request message

The following is a SOAP request message for the Amos II function, *info() -> <Charstring name, Integer age>*:

```
<SOAP-ENV:Envelope
xmlns:SOAP-ENV="http://schemas.xmlsoap.org/soap/envelope/"
xmlns:xsd="http://www.w3.org/2001/XMLSchema"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
<SOAP-ENV:Body SOAP-ENV:encodingStyle="http://schemas.xmlsoap.org/soap/encoding/">
    <tns:INFO xmlns:tns="urn:WSAmos"/>
</SOAP-ENV:Body>
</SOAP-ENV:Envelope>
```

Appendix C: A SOAP response message

The following is a SOAP response message for the Amos II function, *info() -> <Charstring name, Integer age>*.

```
<soapenv:Envelope
        xmlns:soapenv="http://schemas.xmlsoap.org/soap/envelope/"
        xmlns:xsd="http://www.w3.org/2001/XMLSchema"
        xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
        xmlns:SOAP-ENV="http://schemas.xmlsoap.org/soap/encoding/">
<soapenv:Body>
    <tns:INFOReturn xmlns:tns="urn:WSAmos">
        < tns:results>
            < tns:row>
                < tns:NAME xsi:type="xsd:string">Milena</ tns:NAME>
                < tns:AGE xsi:type="xsd:int">38</ tns:AGE>
```

```
            </ tns:row>
            < tns:row>
                < tns:NAME xsi:type="xsd:string">Johan</ tns:NAME>
                < tns:AGE xsi:type="xsd:int">30</ tns:AGE>
            </ tns:row>
            < tns:row>
                < tns:NAME xsi:type="xsd:string">Erik</ tns:NAME>
                < tns:AGE xsi:type="xsd:int">29</ tns:AGE>
            </ tns:row>
        </ tns:results>
    </ tns:INFOReturn>
</soapenv:Body>
</soapenv:Envelope>
```

References

[1]. Staffan Flodin, Martin Hansson, Vanja Josifovski, Timour Katchaounov, Tore Risch, and Martin Sk¨old: *Amos II Release 6 User's Manual*. *UDBL* Technical Report, Dept. of Information Science, Uppsala University, Sweden, March 27, 2004, http://user.it.uu.se/~udbl/amos/doc/amos_users_guide.html

[2] G. Wiederhold, Mediators in the Architecture of Future Information Systems, *IEEE Computer*, 25(3), 1992, pp 38-49

[3]. T.Risch: *Amos II External Interfaces*, UDBL Technical Report, Dept. of Information Technology, Uppsala University, Sweden http://user.it.uu.se/~torer/publ/external.pdf

[4] Elin, D, Risch, T: *Amos II Java Interfaces*, UDBL, Uppsala University, Sweden, August 2000, http://user.it.uu.se/%7Etorer/publ/javaapi.pdf

[5] T.Risch, V.Josifovski: Distributed Data Integration by Object-Oriented Mediator Servers. *Concurrency and Computation: Practice and Experience J.* 13(11), John Wiley & Sons, September, 2001.

[6] *XML Schema Part 1: Structures Second Edition*, W3C Recommendation 28 October 2004. http://www.w3.org/TR/2004/REC-XMLschema-1-20041028/structures.html

[7] *Simple Object Access Protocol (SOAP) 1.1,* W3C Note 08 May 2000. http://www.w3.org/TR/2000/NOTE-SOAP-20000508/

[8] *Web Services Description Language (WSDL) 1.1*. W3C Note 15 March 2001, http://www.w3.org/TR/wsdl

[9] Olaf Zimmermann, Mark Tomlinson, and Stefan Peuser: *Perspectives on Web Services Applying SOAP, WSDL, and UDDI to Real-World Projects*, Springer, 2003, ISBN 3-540-00914-0.

[10] *The introduction of XML in wikipedia*, http://en.wikipedia.org/wiki/XML#Features_of_XML

[11] *Apache Tomcat*, http://tomcat.apache.org/

[12] Frank Cohen: Discover SOAP encoding's impact on Web service performance, http://www-128.ibm.com/developerworks/webservices/library/ws-soapenc/

[13] *Java Web Services, SOAP with Attachments API for Java (SAAJ)*. Sun Microsystems, Inc., http://Java.sun.com/webservices/saaj/index.jsp

[14] *The Universal Description, Discovery and Integration (UDDI) specification* http://www.uddi.org/specification.html

[15] *Web Services Activity,* http://www.w3.org/2002/ws/

[16] *JSoapServer,* http://jsoapserver.sourceforge.net/

[17] *Apache Axis,* http://ws.apache.org/axis/

[18] *QuickServer,* http://www.quickserver.org/

[19] *WSDL4J,* http://sourceforge.net/projects/wsdl4j/

[20] *Namespaces in XML 1.0 (Second Edition)*, W3C Recommendation 16 August 2006, http://www.w3.org/TR/REC-xml-names/

[21] *XML Schema Part 2: Datatypes Second Edition*, W3C Recommendation 28 October 2004, http://www.w3.org/TR/xmlschema-2/

[22] Russell Butek: *Which style of WSDL should I use?* http://www-128.ibm.com/developerworks/webservices/library/ws-whichwsdl/

[23] *The JSR 67 export group,*
http://jcp.org/aboutJava/communityprocess/maintenance/jsr067/index2.html

[24] *SAX (Simple API for XML)*, http://www.saxproject.org/

[25] *DOM (Document Object Model)*, http://www.w3.org/DOM/