Int. J. Communications, Network and System Sciences, 2010, 3, 877-887
doi:10.4236/ijcns.2010.311119 Published Online November 2010 (http://www.SciRP.org/journal/ijcns)

Web Service Generation through Program Slicing*

Yingzhou Zhang[1,2,3], Wei Fu[1,3], Geng Yang[1,2], Lei Chen[1,2], Weifeng Zhang[1]
[1]*College of Computer, Nanjing University of Posts and Telecommunications, Nanjing, China*
[2]*State key Lab. of Networking & Switching Technology, Beijing University of Posts & Telecommunications, Beijing, China*
[3]*State Key Lab. of Novel Software Technology, Nanjing University, Nanjing, China*
E-mail: *zhangyz@njupt.edu.cn*
Received August 21, 2010; *revised September* 25, 2010; *accepted October* 26, 2010

Abstract

As the development of web service (WS), applications based on web services (WS), which are convenient and platform-independent, have become increasingly popular in recent years. However, how to identify, generate and compose services are still open issues. This paper proposes a method based on program slicing to realize the generation and composition of web services. This paper introduces the method about how to generate a WSDL file and a SOAP message from source codes as well as the theory of function dependence graph (FDG). In addition, this paper gives an approach to generate a proxy service for each service, which allows users to easily call a service. The results of experiments show that our generation and composition methods of WS are feasible and flexible.

Keywords: Web Service, Arogram Slicing, WS Generation, Function Dependence Graph, Haskell

1. Background and Related Work

1.1. Program Slicing

Mark Weiser [1] first defines a program slice S as a reduced executable program obtained from a program P by removing statements, such that S replicates part of the behaviour of P. In general, a program slice consists of those statements of a program that may directly or indirectly affect the variables computed at a given program point. The program point p and the variable set V, denoted by <p, V>, is called a slicing criterion. For example, **Figure 1** gives the slicing results for some interest variables of source codes.

Program slicing algorithms can be roughly classified as static slicing and dynamic slicing methods, according to whether they only use statically available information or compute those statements that influence the value of a variable occurrence for a specific program input [2-4]. Program slicing can also be divided into forward slicing and backward slicing [5,6]. In forward slicing, one is interested in what depends on or is affected by the entity selected as the slicing criterion. In backward slicing, one is interested in what affects the variables in the slicing criterion. To help readers understand program slicing more deeply, **Figure 2** gives an example of forward and backward slicing.

```
1   void EgForSlicing ( )
2   {
3       int a = 0;
4       int b = 0;
5       int c = 5;
6       if (c > 10)
7         a ++;
8       else
9         b ++;
10  }
```
(1) source codes

```
1   void EgForSlicing ( )
2   {
5       int c = 5;
10  }
```
(2) static slicing for <10, c>

```
1   void EgForSlicing ( )
2   {
3       int a = 0;
5       int c = 5;
6       if (c > 10)
7         a ++;
10  }
```
(3) static slicing for <10, a>

```
1   void EgForSlicing ( )
2   {
4       int b = 0;
5       int c = 5;
6       if (c > 10)
8       else
9         b ++;
10  }
```
(4) static slicing for <10, b>

Figure 1. Source codes and slicing results.

*This work was supported in part by the National Natural Science Foundation of China (60703086, 60873231, 60873049, 60973046), Jiangsu Natural Science Foundation of China (BK2009426), Qing Lan Project of Jiangsu Province, Jiangsu Innovation Project for postgraduate cultivation (CX10B195Z).


```
1  read (n);                1  read (n);              1
2  j : = 1;                 2  j : = 1;               2
3  sum : = 0;               3                         3  sum : = 0;
4  acc : = 1;               4  acc : = 1;             4
5  while j < = n do         5  while j < = n do       5
6        sum : = sum + 1;   6                         6        sum : = sum + 1;
7        acc : = acc * j;   7       acc : = acc * j;  7
8        j : = j + 1;       8       j : = j + 1;      8
9  write (sum);             9                         9  write (sum);
10  write (acc);            10  write (acc);          10
 (a) source codes             (b) backward slicing      (c) forward slicing
                                  for < 10, acc>            for <3, sum>
```

Figure 2. Examples of forward slicing and backward slicing.

1.2. Web Service

According to the W3C group [7], a WS (also webservice) is traditionally defined as a software system designed to support interoperable machine-to-machine interaction over a network. It has an interface described in a machine-processable format such as WSDL (web services Description Language) [8,9]. Other systems interact with the WS in a manner prescribed by its description using SOAP (Simple Object Access Protocol) messages, typically conveyed through HTTP with an XML serialization in conjunction with other web-related standards.

Web services today are frequently just application programming interfaces (API) or web APIs that can be accessed over a network such as the Internet, and be executed on a remote system hosting the requested services [10].

In general, a service depends on other services. Such dependences can be described as dependence graphs on which program slicing algorithms can be applied for WS generation and composition. All of the algorithms appeared in this paper are implemented in language Haskell which will be introduced in next subsection.

1.3. Haskell

Haskell [11,12] is a lazy, pure functional programming language. It is called lazy because expressions which are not needed to determine the answer to a problem are not evaluated.

Haskell is a functional language because the evaluation of a program is equivalent to evaluating a function in the pure mathematical sense [13-15]. This differs from standard languages (such as C or Java) which evaluate a sequence of statements.

In this paper, we choose Haskell as our implementation language for the reason that it is lazy and pure as it is mentioned above. Haskell possesses an expressive syntax and have rich built-in data types; it is good at handling complex data and combining components. It has lazy semantics: no expression is evaluated until its result is needed. The laziness can encourage the combinatorial and dynamic design of web services. By using Haskell, we can write more bug-free, readable and extensible codes in less time. In addition, Haskell has an advantage of computing mathematical expressions, which is also useful for us to develop WS tools.

1.4. Related Work

Many researchers are doing some related work about WS generation and composition [16,17]. They give some methods and models to solve the problem how to discover and generate a WS. The most two common methods used recently are based on syntax and semantics [18-20].

The method based on syntax is a traditional one. Although this method has been improved by a large number of researchers, it has a known shortcoming when it uses the technology of keyword matching, which often returns some inaccurate results. Many services discovered by this method are not wanted for users.

The method based on semantics does a better job than one which is based on syntax. Its results of discovering services are more accurate. However, there is still a big step to be made to put semantics web services into practical applications for semantics' complex definitions.

Some researchers have studied on generating and composing web services by using program slicing technology, but most of them focus on prepare work for web services, and haven't proposed a feasible method with the whole process of WS generation and composition [21-25]. Rodrigues and Barbosa [26] gave a method for analyzing the relationship between all components in Haskell types/codes. They use program slicing to generate sliced codes for Haskell components. However, they

make less statements in detail about the follow process of WS generation, such as how to generation WSDL documents. What is more, as mentioned in Subsection 1.3, Haskell types/functions are pure and lazy, so their methods of component discovery and function dependence graph can't directly be used for the imperative languages such as C and Java. In this paper, we will make a detailed method about how to generate and compose web services from source codes. The technology of program slicing is used to help us abstract function dependence graph (FDG) and generate corresponding services. Our current prototype of WS tool can cope with the imperative languages C and Java, and the functional language Haskell as well.

The biggest difference between our work and others' is the services' dependences are added when they are generated and published. These dependences make the result more accurate when we do WS discovery for the reason that we can filter the result which is obtained by the discovery through the method of keywords matching. And another difference is that we use program slicing and graph theory to process services' dependences.

The theory of function dependence graph is to be discussed in Section 2, and services identification with program slicing is to be introduced in Section 3. WS generation and composition will be introduced in Section 4. In Section 5, conclusions are drawn and future work is listed.

2. Function Dependence Graph

Program slicing is always based on some dependence graphs such as the control dependence graphs (CFG) and the program dependence graphs (PDG) [26], whose basic node is a statement fragment. However, in our work, function is the basic unit. This will help us to divide source codes into many fragments/components for generating sliced codes of web services. So this section will introduce functional dependency and function dependence graph.

When parsing a source code, a FDG will be generated to record dependences between functions. For simplicity, a function dependence graph is a set, where each element is noted as (a, b), which indicates that the function a depends on the function b and is expressed by an arrow from a to b, *i.e.*, $a \rightarrow b$. For example, **Figure 3** gives a FDG of a source code. In **Figure 3**, each letter in the box stands for one function.

When analyzing the FDG, we need to derivate new elements by doing some operations to elements in the FDG. * Operation is defined to derivate new dependency through some exist dependency in a FDG.

Definition 1: * *operation* is the transitive relation operation which obeys the following rules:

$$(a, b) * (b, c) = (a, c)$$

$$(a, b) * (d, c) = (\text{ø}, \text{ø})$$

$$(a, b) * (\text{ø}, \text{ø}) = (a, b)$$

where a, b, c, d are arbitrary functions such that $b \neq d$, and ø indicates a non-existent function.

In general, the * operation obeys the combination law as follows:

$$((a, b) * (b, c)) * (c, d) = (a, b) * ((b, c) * (c, d))$$

where a, b, c, d are arbitrary functions. However, * operation does not always obey the commutative law.

For example, in **Figure 3**:

$$(x, y) * (y, g) * (g, h) = (x, h)$$

$$(x, y) * (g, h) * (y, g) = (\text{ø}, \text{ø})$$

Therefore, $(x, y) * (y, g) * (g, h) \neq (x, y) * (g, h) * (y, g)$.

Based on the transition operation *, we introduce the follows the direct dependence relation and the indirect dependence relation. If a tuple (a, b) exists in a FDG, the relationship between a and b is the Direct Dependence Relation (DDR), *i.e.*, b is the direct dependent function of a. If a tuple (a, c) does not exist in a FDG, but it can be computed through * operation of several tuples in the FDG, the relationship between a and c is Indirect Dependence Relation (IDR), *i.e.* c is the indirect dependent function of a.

For example, in **Figure 3**, (y, g), (g, h), (g, i) are DDRs and (y, h), (y, i) are IDRs.

In order to apply some slicing methods on FDG for generating web services from source codes, we give two slicing operations. <Operation is the backward slicing operation which begins at the node in a slicing criterion and finds all DDRs recursively>. Operation is the forward slicing operation which can be derived from <operation>. Assuming that c is a slicing node and S is a set of tuples, we have the following equation.

$$c > S = \wedge (c < (\wedge S)) \quad (1)$$

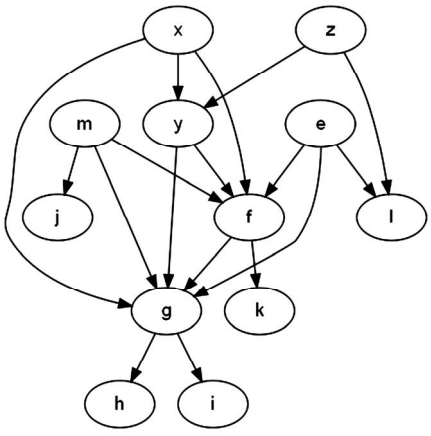

Figure 3. A sample FDG.

Where $\wedge$ operation is inverse relation of a tuple set, *i.e.*

$$\wedge S = \{(b, a) \mid (a, b) \in S\} \quad (2)$$

For example, in **Figure 3**, if y is chosen to be a slicing node and the FDG is noted as S, the result of computation $y < S = \{(y, f), (y, g), (g, h), (g, i), (f, g), (f, k)\}$, and $y > S = \{(x, y), (z, y)\}$.

3. Service Identification with Program Slicing

3.1. Slicing Process

When a server receives some source codes from users, its important work is to identify components/services from the source codes. Non-strictly speaking, each top-level function (or a class in object-oriented languages such as Java) in source codes can be abstracted as a component/service. For simplicity, a global variable or a point variable in C is treated as the special function whose parameter and return are both itself.

For WS identification, source codes will be parsed to build a FDG on which some slicing operations are applied, and then the sliced codes for each service can be easily generated. The main functions of the slicing process will be listed below before describing the algorithm of slicing in details.

code begins:
  codes ← *readFile* (*file*);
  AST ← *generateAST* (*codes*);
  FDG ← *generateFDG* (*AST*);
  SN ← *chooseSlicingNode* (*FDG*);
  FDG′ ← *slicing* (*SN*, *FDG*);
  AST′ ← *compare* (*FDG′*, *AST*);
  slicedCodes ← *writeCodes* (*AST′*)
code ends

Figure 4 shows the slicing process framework of WS identification from source codes. The slicing process can be divided into five steps below.

***Step* 1.** Produce the abstract syntax tree (AST [26]) for a source code.

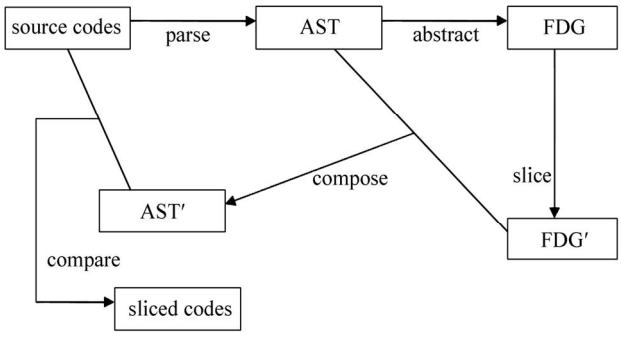

Figure 4. Program slicing framework for WS identification.

For convenience, the data type Method is defined as follows to record information about a function.

data *Method* = (*Name, Paras, Return, Location, Method*)

The Method type has five child types: Name, Paras, Return, Location, Method, which records the current method name, its parameters' name and types, the return type, the location information, and the other method type in its DDRs of the functions respectively. The Method type is a recursive type for the reason that a Method type would exist in another Method type.

We will generate an instance of the Method type for each function in source codes. After finishing parsing the codes, many instances of the Method type can be generated.

***Step* 2.** Generate a FDG from the AST of source codes.

A FDG is in fact a set of function tuples. FDG can be generated as long as this set can be obtained from the Method type. As mentioned above, the Method type is a recursive type, so a recursive algorithm is descried into two steps to process an instance of the Method type. The first step is processing parent instance of the Method type and the second step is processing the child instance of the Method type of the parent instance.

Step 2.1: checking whether the value of child Method in a parent Method is NULL. If the value is NULL, no tuple will be generated, and if not, a tuple (Method.Name, Method.Method.Name) will be generated.

Step 2.2: assigning parent Method's value with child Method's value and implement **Step 2.1** again.

The follows gives the corresponding function code of processing an instance of a Method type.

tuple Method =
  if (*Method.Method == NULL*)
    then *NULL*;
    else *return* (*Method.Name*, *Method.Method.Name*);
  tuple Method.Method

A set of function tuples can be abstracted from AST through processing all instances of the Method type. Then, the FDG can show all function dependences of a source code.

***Step* 3.** Implement slicing on a FDG at a slicing node.

A simple way to fulfill this step is to implement <operation and> operation on a FDG.

The algorithm of this step is also a recursive one and divided into two steps. The first step is to find DDRs of a slicing node and the second one is processing the DDRs found in the first step.

Step 3.1: finding all direct dependent functions of a slicing node.

Step 3.2: If the result functions in **Step 3.1** is NULL, the algorithm will come to an end. Otherwise, its direct dependent function will be treated as a new slicing node

and **Step 3.1** will be implemented again.

The corresponding function code of ***Step 3*** is listed below.

slicing (SN, FDG) =
{ *SN′ ← getDirectDependence (SN, FDG);*
if *SN′ == NULL* ***then*** *NULL;*
else *slicing (SN′, FDG)* }

where SN is a slicing node.

In this paper, we focus on backward slicing because the generation of sliced codes is based on the result of implementing backward slicing from a slicing node. After ***Step* 3**, a FDG′ will be generated.

***Step* 4.** Pruning the AST of source codes according to the FDG′ obtained in ***Step* 3**.

After ***Step* 3**, we can obtain all the names of the nodes in the FDG′. We then decide to remain or delete a method name in AST according to whether or not it is in the FDG′. Its implementation function code can be found below.

compare (FDG′, AST) =
{ *nodes ← getAllNodes FDG′;*
Method ← getASTMethod AST;
again: ***if*** *(search (Method , nodes) == False)*
then *delete Method;*
Method = Method → next;
Goto again;
else *remain Method;*
Method = Method → next
Goto again }

***Step* 5.** Generate sliced codes from source codes according to the AST′ from ***Step* 4**.

The algorithm of this step is rather easy because the location of a function in the source codes can be obtained from the Method.Loacation type. The only work to do is to abstract functions in the AST′ from source codes and put them together into sliced codes.

Figures 5-7 below show the results of a slicing process. With our WS toolkit prototype, **Figure 5** shows the sample C code in its left frame, whose FDG is shown in **Figure 6**, and the sliced code for the function "fun3" in its right frame, whose FDG is shown in **Figure 7**. As mentioned at the beginning of this section, we treat a global variable as a particular function whose parameter and return are both itself. So, in order to distinguish these particular functions, we here adopt the dotted line as shown in **Figures 6** and **7**.

In fact, our current WS toolkit written in Haskell can be carried out to do slicing process for source codes of C, Java and Haskell, whose main interface is shown in **Figure 8**.

There are two frames in **Figure 8** for displaying codes. The left frame shows source codes and the right one shows sliced codes. In **Figure 8**, for example, a Haskell

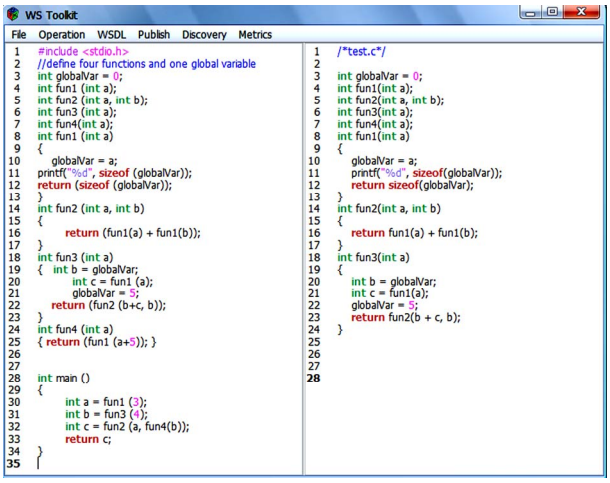

Figure 5. A C source code (in left column) and its sliced code for "fun3" (in right column).

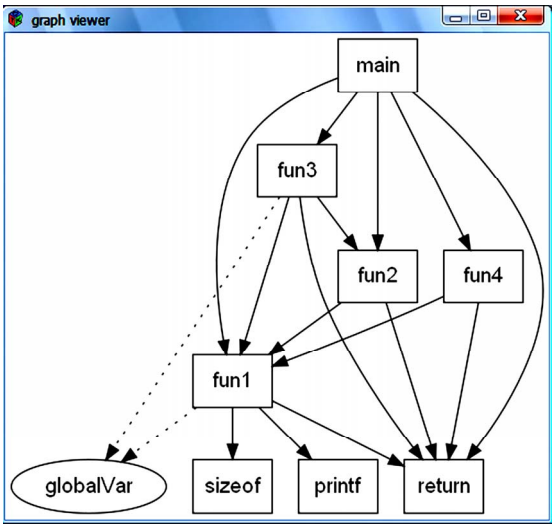

Figure 6. The FDG of the source code in Figure 5.

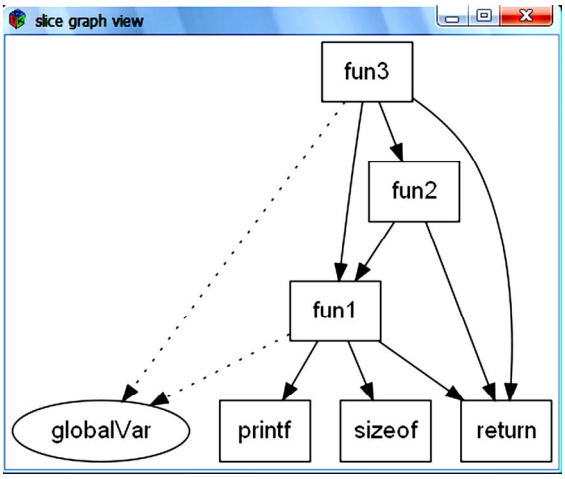

Figure 7. The FDG′ of slicing for "fun3" code in Figure 6.

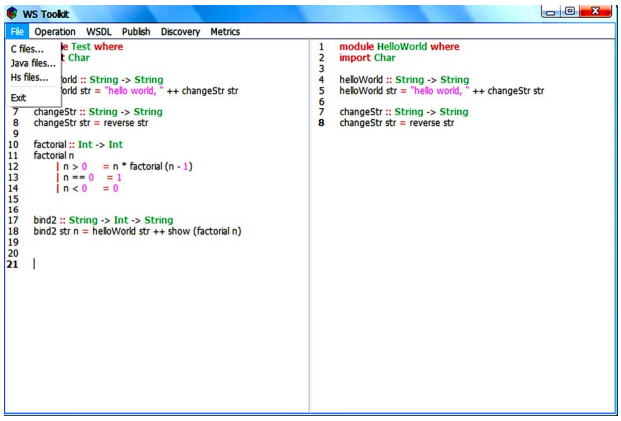

Figure 8. The main interface of our WS toolkit.

source code is showed in the left frame and a sliced code is showed in the right frame. The sliced code is result of backward slicing from the slicing node "helloWorld". From the sliced code, it can be seen that function "helloWorld" depends on function "changeStr". Therefore, both the definitions about function "helloWorld" and function "changeStr" have been remained in the sliced codes.

As showed in above examples, some functions such as auxiliary functions in source codes, may not have good interface to be published as a WS. Program slicing technology can compose several services and generate their codes, but it has no ideal method to recognize functions which have good interface. How can we recognize such functions? In fact, users who upload source codes know it clearly. Therefore, it is helpful when users give a WSDP (Web Service Description for Publishing) file to describe which functions in a source code to be published as web services.

3.2. WSDP Files

A WSDP (Web Service Description for Publishing) file, which states what functions to be published as web services, is written in the XML file format so that it can be easily embedded into WS framework.

Before slicing the source codes, WSDP files should be parsed to obtain some useful information. The data type WSDPInfo is defined as follows to record the useful information in a WSDP file.

data *WSDPInfo* =

(*MainFileName, Authors, Operator, FuncName, FuncDoc*)

Where the types MainFileName, Authors and Operator describe the basic information of a source file. The type FuncName records the names of functions wanted to be published as a service and the FuncDoc describes some information of each service.

An example of a WSDP file can be found in **Figure 9**. Through parsing the WSDP file in **Figure 9**, we can know those functions' names that will be published as services and some other useful information.

As is well known, the WSDL document of a service is very useful when the service is published. Based on the slicing process and result, we discuss in next section to generate a WSDL document for each service to be published.

4. WS Generation and Composition

4.1. Generation of a WSDL Document

WSDL is an XML-based language that provides a model for describing web services. It describes the location and the name of a service as well as the operation or function the service provides. It contains a series of definitions about a service. Some major elements of a WSDL document are listed in **Table 1**.

Figure 10 shows a simple example of a WSDL document. In **Figure 10**, the <portType> element defines "Version" as a port name, "getVersion" as an operation name. The operation "getVersion" has an input message named "getVersionRequest" and an output message named "getVersionResponse".

In order to generate a WSDL document more conveniently, we define as follows the date type Opr_Info to record the content of the important elements in a WSDL document.

```
<?xml version="1.0" encoding="UTF-8"?>
<serviceDetail>
    <mainFileName>Test.hs</mainFileName>
    <authors>
        <author>Wei Fu</author>
        <author>Yingzhou Zhang</author>
    </authors>
    <operator>NJUPT</operator>
    <serviceFunctions>
        <serviceFunction>
            <serviceFuncName>helloWorld</serviceFuncName>
            <serviceFuncDoc>This is a hello world test.</serviceFuncDoc>
        </serviceFunction>
        <serviceFunction>
            <serviceFuncName>factorial</serviceFuncName>
            <serviceFuncDoc>This is a factorial test.</serviceFuncDoc>
        </serviceFunction>
        <serviceFunction>
            <serviceFuncName>bind2</serviceFuncName>
            <serviceFuncDoc>This is binding test of two tests.</serviceFuncDoc>
        </serviceFunction>
    </serviceFunctions>
</serviceDetail>
```

Figure 9. An example of a WSDP file.

Table 1. The main elements of a WSDL document.

ELEMENT	DEFINITION
<portType>	the operation of a WS
<message>	the messages used by a WS
<binding>	the transportation protocol
<service>	the detail and important information

data *Opr_Info*
= (*Opr_InN, Opr_InT, Opr_OutT, Opr_Name*)

The type Opr_Info has four child types: Opr_InN (input parameter names), Opr_InT (input parameter types), Opr_OutT (return value's type) and Opr_Name (the name of an operation).

The type Opr_Info can be obtained through parsing source codes, and **Figure 11** shows the process of generating elements <message> and <portType> from type Opr_Info.

For example, if there is a function named "hello" below to be published as a WS.

```
String hello (String name)
{
        printOutString ("welcomg:"+ name);
}
```

The corresponding Opr_Info type for the function "hello" is: Opr_Info = {Opr_InN = name, Opr_InT = String, Opr_OutT = String, Opr_Name = hello}. The WSDL elements <message> and <portType> of "hello" function are shown in **Figure 12**.

The WSDL element <binding> describes the related protocols and the format of the messages for each port. It tells a server which styles are used to transfer messages between the server and a client. **Figure 13** shows the WSDL <binding> of the example function "hello" above. Here we choose RPC (Remote Procedure Call) as the transmission style.

The WSDL element <service> describes the detail information of a service, such as its name and web address. For example, supposing the server address of the "hello" function is "http://localhost:8080/axis/Hello.jws", we will get such WSDL <service> descriptions as shown in **Figure 14**.

Since the generations of the WSDL element <message>, <portType>, <binding>, <service> has been introduced above, the generation of a WSDL document would be very easy by composing them together.

Our WS toolkit has a "WSDL" menu to generate WSDL documents for the functions to be published. For the example code in **Figure 8**, the WSDL menu in **Figure 15** has three submenu items named "helloWorld", "factorial" and "bind2", which are chosen by its WSDP file in **Figure 9**. Since the WSDP file does not list "changeStr" as a service to be published, the WS toolkit only generates WSDL documents for these three functions. **Figure 16** gives the result of the WSDL document generated for the "helloWorld" function.

```
<wsdl:message name="getVersionRequest"/>
<wsdl:message name="getVersionResponse">
    <wsdl:part name="getVersionReturn" type="xsd:string"/>
</wsdl:message>
<wsdl:portType name="Version">
    <wsdl:operation name="getVersion">
        <wsdl:input name="getVersionRequest" message="impl:getVersionRequest"/>
        <wsdl:output name="getVersionResponse" message="impl:getVersionResponse"/>
    </wsdl:operation>
</wsdl:portType>
```

Figure 10. An example of a WSDL document.

```
<wsdl:message name="Operation_NameResponse">
    <wsdl:part name="Operation_NameReturn" type="xsd:Operation_OutputT"/>
</wsdl:message>
<wsdl:message name="Operation_NameRequest">
    <wsdl:part name="Operation_InputN " type="xsd:Operation_InputT"/>
</wsdl:message>
<wsdl:portType name="Service">
    <wsdl:operation name="Operation_Name " parameterOrder="Operation_InputN">
        <wsdl:input name="Operation_NameRequest" message="impl:Operation_NameRequest"/>
        <wsdl:output name="Operation_NameResponse" message="impl:Operation_NameResponse"/>
    </wsdl:operation>
</wsdl:portType>
```

Figure 11. The part of a WSDL document generated from an Opr_Info type.

```
<wsdl:message name="helloResponse">
    <wsdl:part name="helloReturn" type="xsd:string"/>
</wsdl:message>
<wsdl:message name="helloRequest">
    <wsdl:part name="name" type="xsd:string"/>
</wsdl:message>
<wsdl:portType name="Hello">
    <wsdl:operation name="hello" parameterOrder="name">
        <wsdl:input name="helloRequest" message="impl:helloRequest"/>
        <wsdl:output name="helloResponse" message="impl:helloResponse"/>
    </wsdl:operation>
</wsdl:portType>
```

Figure 12. The WSDL <message> and <portType> of the example function "hello".

4.2. Generation of SOAP Messages

Simple Object Access Protocol (SOAP) [27-29] is a simple XML-based protocol that allows applications to exchange information over HTTP. Simply speaking, SOAP is a protocol which is used to access web services, and its message is simple XML documents.

RPC [30] is now used by current applications such as DCOM and CORBA to communicate with each other.

```
<wsdl:binding name="HelloSoapBinding" type="impl:Hello">
    <wsdlsoap:binding style="rpc"
    transport="http://schemas.xmlsoap.org/soap/http"/>
    <wsdl:operation name="hello">
        <wsdlsoap:operation soapAction=""/>
        <wsdl:input name="helloRequest">
            <wsdlsoap:body use="encoded"
            encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"
            namespace="http://DefaultNamespace"/>
        </wsdl:input>
        <wsdl:output name="helloResponse">
            <wsdlsoap:body use="encoded"
            encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"
            namespace="http://localhost:8080/axis/Hello.jws"/>
        </wsdl:output>
    </wsdl:operation>
</wsdl:binding>
```

Figure 13. The WSDL <binding> of the "hello" function.

```
<wsdl:service name="HelloService">
    <wsdl:port name="Hello" binding="impl:HelloSoapBinding">
        <wsdlsoap:address location="http://localhost:8080/axis/Hello.jws"/>
    </wsdl:port>
</wsdl:service>
```

Figure 14. The WSDL <service> of the "hello" function.

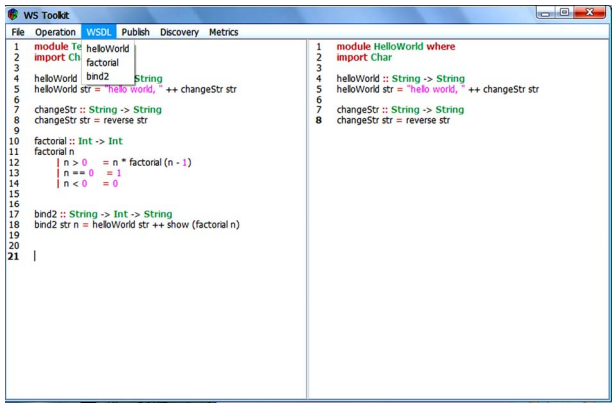

Figure 15. The WSDL menu in the WS toolkit.

Figure 16. The result of generating WSDL for "helloWorld" from our WS toolkit.

However, it's not designed for HTTP. Because of compatibility and security issues, RPC is always be blocked by firewall and proxy servers.

An alternative way to communicate between applications is using HTTP with supporting by almost all browsers and servers. Based on HTTP, SOAP provides some standard functions to make applications communicate to each other, so it make web services more heterogeneous than web applications. Web applications use multiple programming languages, but web services can also use different operating systems, different server containers.

When users want to use a published service, they can send requests to a server through SOAP messages. A SOAP message contains the operation name and related parameter values of a service. It can be generated by parsing the WSDL document of a service. For simplicity, the SOAP messages instantiated in this paper only contains two required elements <envelope> and <body> shown in **Table 2**.

Figure 17 shows the relationship between WSDL and SOAP with three document fragments. Fragment 1 is the <message> and <portType> elements of a WSDL document. Fragment 2 is a SOAP message, and Fragment 3 is the <operation> element of a WSDL. The arrows indicate how to use a WSDL document to generate a SOAP message. For example, names and types of the parameters in <body> element of Fragment 2 come from the <message> element of Fragment 1.

Figure 18 shows the process about how to generate a SOAP message from a WSDL document.

In practice, we encapsulate the information a SOAP message need into a data type SOAP_Info when parsing a WSDL document.

data *SOAP_Info*
= (*In_Para, Out_Para,Opr_Name, In_Ns, Out_Ns*)

Table 2. The main elements of a SOAP message.

ELEMENT	ATTRIBUTE	FUNCTION
<envelope>	REQUIRED	implies this is a SOAP message
<header>	OPTIONAL	contains some header messages
<body>	REQUIRED	contains request and response messages
<fault>	OPTIONAL	provides messages dealing with errors

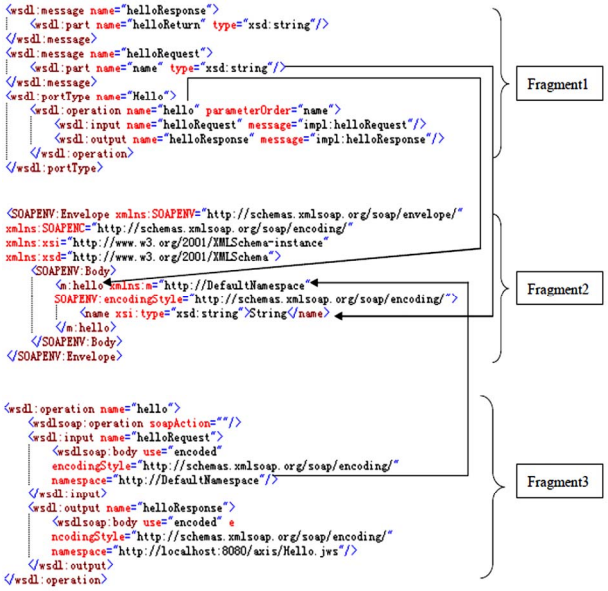

Figure 17. Relationship between WSDL (Fragment 1 and Fragment 3) and SOAP (Fragment 2).

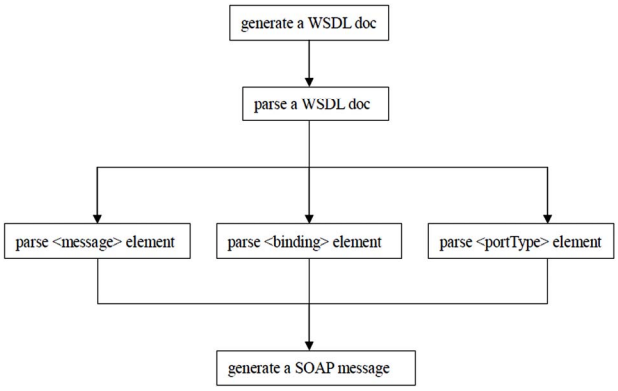

Figure 18. The process of generating a SOAP message from a WSDL document.

The SOAP_Info type has five child types. In_Para (or Out_Para) records the parameters' names and types of a service's input (or return value). Opr_Name describes the names of the operations in a SOAP message. The In_Ns/Out_Ns type is related with the namespace of a SOAP request/response. Based on the SOAP_Info type, a SOAP message can be generated and then be sent to the related WS.

4.3. Procedure of SOAP Transmission

In this subsection, we will discuss how to send the SOAP message generated from a WSDL document to a server. As is known that SOAP is based on HTTP, we just need to seed a HTTP packet in which a SOAP message is encapsulated to the server related.

Before sending SOAP messages to a server, the server's location address, which is usually called endpoint, must be known. This endpoint can also be analyzed from WSDL documents by parsing the <service> element.

The element <service> describes some detail and important information of a service. For example, from **Figure 19**, we know that the element <service> describes the name and the port of a service as well as the location address. So from the WSDL element <service>, we can easily obtain the location address (endpoint) of the transmission destination of a SOAP message.

Once a server receives a request SOAP message, the server will generate a response SOAP message with the result of calling the specific operations.

For example, **Figures 20-21** give a request and its response SOAP message respectively. For simplicity, we here only show the elements <body> of SOAP messages, and omit their other elements such as <envelope>. There's no information about input parameters in **Figure 20** because the "getVersion" operation doesn't needs any parameter.

Furthermore, **Figure 22** shows the procedure of sending and receiving SOAP messages, each of which is packaged into a HTTP packet when transmitted between users and servers.

When calling an operation of a service, an important function "runService" below is used to send HTTP packets to servers, with passing parameters to the operation of a service and returning its result values. Through such operations, communication between users and server can be established.

runService request =
{
Request request;
-- package a SOAP message, named request

```
<wsdl:service name="VersionService">
  <wsdl:port name="Version" binding="impl:VersionSoapBinding">
    <wsdlsoap:address location="http://localhost:8080/axis/services/Version"/>
  </wsdl:port>
</wsdl:service>
```

Figure 19. A <service> element of a WSDL document.

```
<SOAPENV:Body>
  <m:getVersion xmlns:m="http://axis.apache.org"
  SOAPENV:encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"/>
</SOAPENV:Body>
```

Figure 20. An example request SOAP message.

```
<soapenv:Body>
  <ns1:getVersionResponse soapenv:encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"
   xmlns:ns1="http://axis.apache.org">
    <getVersionReturn xsi:type="xsd:string">
      Apache Axis version: 1.4 Built on Apr 22, 2006 (06:55:48 PDT)
    </getVersionReturn>
  </ns1:getVersionResponse>
</soapenv:Body>
```

Figure 21. An example response SOAP message.

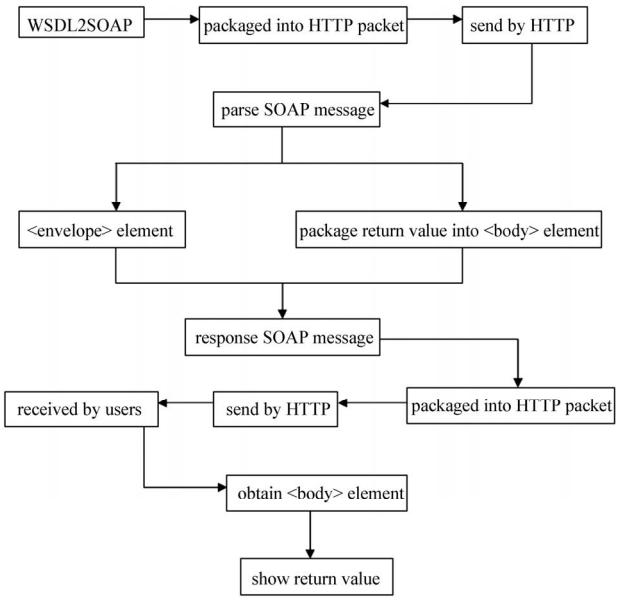

Figure 22. The procedure of SOAP transmission.


```
    result = simpleHTTP request operation;
           -- use the simpleHTTP function to
           -- send request package
  case result of
      error → putout ("service error"+ error);
      right soap → case soap of
           -- if right, return response SOAP message
           -- and parse it
               error → putout ("soap error"+ error);
               right body → getbodyContent body;
                  -- if right, get the body content
               }
```

4.4. Proxy Services

In practical applications, to automatically run the above functions of generating WSDL document and SOAP messages, we often introduce the function of a proxy service. If such a proxy service has been developed, users only need to provide the parameters of a service operation to its proxy service, and the server will automatically call related functions to generate WSDL documents and SOAP messages with the result of the service operation. To ensure that the proxy service can work properly, all of the data transmission and display should be based on XML. So the types of a service operation must be converted into XML-format types.

There is only one data type named string in XML since XML is based on text. Therefore, all data types of a service must be converted into an XML-format type. To fulfill such requirement, a class is defined as follows.

```
class Xmlable a where
   toContent :: a → [[Content]]
   fromContent :: [[Content]] → a
```

In the Xmlable class, there are two operations: toContent and fromContent, which aims at converting an arbitrary type, a, into an XML-format type, and restoring an arbitrary type form a non-original XML format respectively. The toContent function can automatically convert the operation parameters of a service into XML-format as long as it is declared as an instance of Xmlable.

In fact, the function "runService" mentioned in Subsection 4.3 is a prototype of proxy service. It has only one parameter named SOAP_Info which can be generated form a SOAP request message. Prefix "ws_" is added to a service name as its proxy service's name. All data types of a service must be declared as instances of class Xmlable when generating a proxy service for a service. Servers will call the proxy service directly and get the operation's return value.

A simple example of generating a proxy service can be found below.

```
hello:: String → String
hello str = "welcome" ++ str
   -- Server generates a proxy service name ws_hello:
ws_hello soapinfo = runService soapinfo

Instance Xmlable String
 -- declare type String as an instance of Xmlable
```

5. Conclusions & Future Work

5.1. Conclusions

This paper introduces the method about how to generate and compose services using program-slicing technology based on function dependence graph. Through parsing source codes, sliced codes for each service can be generated. It also introduced the method about how to generate WSDL documents and SOAP messages through sliced codes of each service.

To make sure web services in a server work smoothly, we generate a proxy service/function for each service. Users can call a WS by accessing the proxy function of that service. In addition, we have introduced the SOAP transmission between users and servers.

5.2. Future Work

This paper has proposed how to generate and compose services and how to generate WSDL documents and SOAP messages. However, no mechanism about how to publish a service is introduced. Users who want to access a service must know the endpoint of the service. But we haven't distributed an endpoint for each service on a HTTP server. Therefore, an important part of our future work is to publish a service on servers.

Another part of our future work is to study BPEL4WS (Web Services Business Process Execution Language for Web Service, which is also based-on-XML language) [31], and to generate corresponding operation dependence graph from a BPEL4WS file. BPEL4WS specifies interactions with web services. Through an operation dependence graph, we can verify the consistency of the composition of some services and evaluate their composition quality.

6. References

[1] M. Weiser, "Program Slicing," *IEEE Transactions on Software Engineering*, Vol. 16, No. 5, 1984, pp. 498-509.

[2] F. Tip, "A Survey of Program Slicing Techniques," *Journal of Programming Languages*, Vol. 3, No. 3, 1995, pp. 121-189.

[3] D. Binkley and K. B. Gallagher, "Program Slicing," *Advances in Computers*, Vol. 43, 1996, pp. 1-50.

[4] Y. Zhang and B. Xu, "A Novel Formal Approach to Pro-

gram Slicing," *Science in China: Information Sciences*, Vol. 50, No. 5, 2007, pp. 657-670.

[5] B. Jon and E. David, "Program and Interface Slicing for Reverse Engineering," *Proceedings of the* 15*th International Conference on Software Engineering*, 1993.

[6] J. Boye, J. Paakki and J. Mduszyfy, "Synthesis of Directionality Information for Functional Logic Programs," In Third International Workshop WSA'93, *Proceeding Static Analysis*, Padova, Vol. 724, 1993, pp. 165-177.

[7] Web-Service website, 2009. http://en.wikipedia.org/wiki/Web_service

[8] F. Curbera, M. Duftler, R. Khalaf, W. Nagy, N. Mukhi and S. Weerawarana, "Unraveling the Web Services Web: An Introduction to SOAP, WSDL and UDDI," *IEEE Internet Computing*, Vol. 6, No. 2, April 2002, pp. 86-93

[9] WSDL website: http://en.wikipedia.org/wiki/Web_Services_Description_Language

[10] K. Yue, X. Wang and A. Zhou, "Underlying Techniques for Web Services: A Survey," *Journal of Software*, Vol. 15, No. 3, 2004, pp. 428-441.

[11] H. Daume III, "Yet Another Haskell Tutorial," Technical Reports, University of Utah, 2003. http://www.cs.utah.edu /~hal/htut/tutorial.pdf

[12] G. Hutton, "Programming in Haskell," Cambridge University Press, New York, 2006.

[13] Y. Zhang and B. Xu, "A Novel Formal Approach to Program Slicing," *Science in China Series F: Information Sciences*, Vol. 50, No. 5, 2007, pp. 657-670.

[14] J. Backus, "A Functional Style and Its Algebra of Programs," *Communications of the ACM*, Vol. 21, No. 8, 1978, pp. 613-641.

[15] J. Peterson, K. Hammond and L. Augustsson, "A Non-Strict Purely Functional Language," Yale University Technical Report, YALEU/DCS /RR, pp 1106- 1997.

[16] M. Feng, "Review of Web Services Composition," *Computer Applications and Software*, Vol. 124, No. 2, February 2007, pp. 23-27.

[17] W. Ou, K. Wang, C. Zeng, D. Li and Z. Peng, "Framework of Multi-Source Web Service Discovery," *Journal of PLA University of Science and Technology*, Vol. 9, No. 5, October 2008, pp. 431-435.

[18] L. Ye and B. Zhang, "A Method of Web Service Based on Functional Semantics," *Journal of Computer Research and Development*, Vol. 44, No. 8, 2007, pp. 1357- 1364.

[19] Z. Zhang, C. Zuo and Y. Wang, "Web Service Discovery Method Based on Semantic Expansion," *Journal on Communications*, Vol. 28, No.1, January 2007, pp. 57-63.

[20] J. Wu, Z. Wu, Y. Li and S. Deng, "Web Service Discovery Based on Ontology and Similarity of Words," *Chinese Journal of Computers*, Vol. 28, No. 4, April 2005, pp. 694-702.

[21] Graph Type website: http://hackage.haskell.org/package/graphtype

[22] F. Liu, Q. Tan and Y. Yang, "Graph-Based Algorithm of Web Services Composition," *Journal of Huazhong University of Science and Technology* (*Nature Science*), Vol. 33, 2005, pp. 202-204.

[23] L. Cao, J. Liu and H. Miao, "Study on Optimization of Web Services Composition Based on Graph," *Chinese Journal of Computers*, Vol. 34, No. 2, 2007, pp. 95-99

[24] S. Deng, J. Yin, Y. Li, J. Wu and Z. Wu, "A Method of Semantic Web Service Discovery Based on Bipartite Graph Matching," *Chinese Journal of Computers*, Vol. 31, No. 8, 2009, pp. 1364-1374.

[25] O. Claudio and G. Josep Silva, "A Slicing Tool for Lazy Functional Logic Programs," Technical Report, University of Madrid, 2006.

[26] F. Rodrigues and S. Barbosa, "Component Identification Through Program Slicing," *Proceeding of Formal Aspects of Component Software* (*FACS* 2005), pp. 231-245.

[27] D. Weaire and D. Rivier, "Soap, Cells and Statistics-Random Patterns in Two Dimensions," *Contemporary Physics*, Vol. 50, No. 1, 2009, pp. 199-239.

[28] J. Peter and M. Albert, "The Soap Chamber Test: A New Method for Assessing the Irritancy of Soaps ," *Journal of the American Academy of Dermatology*, Vol. 1, No. 1, July 1979, pp. 35-41.

[29] SOAP website, 2009. http://en.wikipedia.org/wiki/SOAP

[30] S. Simon, D. Edd and J. Joe, "Programming Web Services with XML-RPC," O'Reilly and Associates, 2001.

[31] BPEL website , 2009. http://en.wikipedia.org/wiki/Business_Process_Execution_Language

