Computing and Informatics, Vol. 32, 2013, 968–986

AN EXECUTABLE SERVICE COMPOSITION CODE AUTOMATIC CREATION TOOL BASED ON PETRI NET MODEL

Zhijun Ding, Jieqi Liu, Junli Wang[*]

Department of Computer Science and Technology
Tongji University
Shanghai 201804, China
e-mail: `zhijun_ding@outlook.com, junliwang@tongji.edu.cn`

Fang Wang

Department of Information Systems and Computing
Brunel University
Uxbridge UB8 3PH, United Kingdom
e-mail: `fang.wang@brunel.ac.uk`

Communicated by Maozhen Li

Abstract. For Web services composition problem, this paper proposes an executable code creation algorithm to model Web services composition based on Petri net model, and develops an executable composition code automatic creation tool. This tool can achieve the automatic creation process from composition model to executable code, and more meaningfully makes it possible to analyze and validate composition process logically. Finally, experiment results have proven that the tool of this paper is feasible.

Keywords: Petri net model, web services composition, executable code

Mathematics Subject Classification 2010: 68-N19

[*] corresponding author

1 INTRODUCTION

As functional requirements in reality become more and more complex, user's demands often could not be satisfied with only one single Web service. Since that, there should be a possibility to combine some existing services together for fulfilling a more complex function request, that is, the Web services composition (WSC) problem comes [1]. There are some standard Web services protocols such as SOAP, WSDL, UDDI and so on, which support flexible interactions between Web services with one accord; and, if the process of composition between Web services can be achieved by GUI-based tool automatically, the efficiency of developing Web services composition will be highly promoted. The tools such as WebSphere Business Integration in IBM [2] and BPEL PM Designer in Oracle [3] use the method of process graph to describe the process of Web services composition, create a special BPEL document, publish the BPEL document to special engine(for example, ODE engine), and at last, run the process composition on a server. On the Protégé tool [4] that is developed by Stanford University, a visualization tool of OWL-S based services composition process logically divides services composition process into a series of standard modules, such as sequence, split, split-join, repeat and so on. This tool transforms the logical composition process of Web services into the augment process of Web services composition process with standard module. All these aforementioned tools make different improvement for Web services composition process in many different aspects; but, due to the lack of formal models, they cannot support the analysis of some important crucial properties for the correctness of the composition logic, such as reachability, deadlock and so on.

Petri net is a formalization method that is well suitable for analysis and verification of distribution systems, and there are many researches on simulation of Web Services composition with Petri net [5, 6].

Narayanan and Mcllraith use Petri net as a tool for model construction, simulation and analysis of DAML-S markups, and then verify the correctness and validity of a composite Web service with reachability analysis method of Petri net [7]. Ding et al. present a hybrid approach for synthesis Petri nets for modeling and verifying composite Web Service, and this approach can be used to validate the correctness or soundness of Web services composition [8]. Hamadi and Benatallah propose a Petri net-based algebra for modeling Web services control flows, and the model is constructed by step-by-step net refinement, which can support hierarchical modeling for composite service [9].

However, these researches are merely based on theory, and some key problems still remain unsolved, such as how to model a Web services composition with Petri net, and how to realize a development tool of Web services composition. For studying and demonstrating an automatic code creation method of Web services composition, we presented a Petri net-based automatic executable code generation method for WSC [10]. This paper is full extension of [10]. Beside including Petri net model for WSC and an automatic WSC executable code generation algorithm as stated in [10], this paper develops a Web services composition code creation tool and presents system architecture and realization process of the tool, in detail. Moreover, a practical WSC example runs throughout the whole article to illustrate our method and tool.

The rest of this paper is organized as follows. Section 2 gives an instance of Web services composition as the whole-length example. Some related definitions of the Petri net model for Web services composition are introduced in Section 3. Based on the constructed Petri net model for services composition, Section 4 presents an algorithm to automatically generate executable Web services composition code. Section 5 develops a Petri net based Web services composition code creation tool, describes the tool's framework and some necessary pretreatment works about composing Web service, and then lays out detailed process from Petri net model to executable code for the example. Finally, the conclusion and future work are presented in Section 6.

2 ILLUSTRATING EXAMPLE

Suppose a situation in which a Web application wants to get local weather information in English according to a machine's IP address. This application is a complex one, because generally there is no such independent Web service on the Web to fulfil this function; but through some analysis, this issue can be figured out by three sub-issues as follows:

1. how to automatically get current location of the machine;
2. how to get the location's weather;
3. how to translate the weather information into English.

After subdivision of the complex application into some simpler applications, it is easier to find their suitable Web services on the Web shown as follows [9] which figure out the three above issues:

1. Web service *IP2Address*: queries geographical location according to machine's IP;
2. Web service *Weather*: queries weather report in Chinese using geographical location;
3. Web service *Translation*: translates weather report in Chinese into English.

Web Service	**Input**	**Output**
IP2Address	IP address	Location of the machine
Weather	Location	Weather information in Chinese
Translation	Weather information in Chinese	Weather information in English

Table 1. Web services and their input/output interfaces

Since every machine on the Internet has its own unique IP address for a given machine, according to its IP address we can get its geographical location by *IP2Address* service whose input is IP address and output is geographical location. Then, we can use the geographical location from *IP2Address* as *Weather* service's input and get the weather information in Chinese by service *Weather.* At last, we use the Chinese weather information from service *Weather* as the input of *Translation* service to get the final information, the local weather information of the machine in English.

The above process of composing several simple services to fulfil a complex task is just a basic process of Web services composition. In a complex Web services composition logic, some composite Web service's properties, such as reachability, deadlock, dead-circulate and so on, are difficult to be analyzed manually. Hence, it is very necessary to build a formal model for analyzing and validating these properties automatically.

3 THE PETRI NET MODEL OF WEB SERVICE

3.1 Petri Net

A net is a quadtuple $N = (P, T, F, W)$, where P is a finite set of places, T is a finite set of transitions such that $P \bigcap T = \emptyset$ and $P \cup T \neq \emptyset$, $F \subseteq (P \times T) \cup (T \times P)$ is the flow relation and W is a weight function such that if $(x, y) \in F$, $W(x, y) \in \mathbb{N}^+$(the set of positive integers), otherwise, $W(x, y) = 0$. A net is said to be ordinary if $W(x, y) = 1$, $\forall (x, y) \in F$. In this case, W is omitted. The pre- and post-sets of a node $x \in P \cup T$ are defined as ${}^\bullet x = \{y \in P \cup T | (y, x) \in F\}$ and $x^\bullet = \{y \in P \cup T | (x, y) \in F\}$, respectively.

The marking (or state) of a net is a function $M : p \rightarrow \mathbb{N}$ (the set of non-negative integers), represented by a multi-set expression or a $|P|$-vector $(M(p_1), \ldots, M(p_{|P|}))^T$, where $M(p)$ is the number of tokens in place $p \in P$. A Petri net $PN = (N, M_0)$ is a net N with an initial marking M_0. A transition $t \in T$ is said to be k-enabled for $k \in \mathbb{N}^+$ at marking M iff $\forall p \in {}^\bullet t : M(p) \geq k \cdot W(t, p)$. If t is 1-enabled, it is denoted as $M[t >$. Firing an enabled transition t results in changing M into M' represented by $M[t > M'$, where $\forall p \in P$, $M'(p) = M(p) - W(p, t) + W(t, p)$. A sequence of transitions $\sigma = t_1 \cdot t_2 \ldots t_k$ is a firing sequence if there exists a sequence of markings such that $M[t_1 > M_1[t_2 > \ldots M_k$, it can be written as $M[\sigma > M_k$, and M_k is said to be reachable from M by firing σ. For more definition and terminology of Petri nets refer to [11].

3.2 The Definition of Petri Net Model for Web Services Composition

In the Petri net model of Web services composition, we use transition to represent the Web service that will be invoked, place to represent data's source or target, and token to represent the data value assigned to Web service. Each place in this paper has zero or one token. For example, for the *IP2Address* service in our example, a Petri net model can be obtained as shown in Figure 1. In the *IP2Address*'s Petri net model, place *IP* holds the data that *IP2Address* service needs, and place *address* holds the data that *IP2Address* service creates. Transition *IP2Address* represents the execution of *IP2Address* service.

At the same time, for creating executable code by Petri net model, some types of related information are collected by adding attributes of elements of Petri nets, which will be introduced and explained in the sequel.

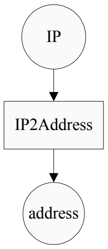


Figure 1. Petri net model of *IP2Address* service

3.2.1 Place

Place denotes data's source or target, and has five attributes: *placeName*, *isALiveToken*, *tokenIsTo*, *inputList*, *outputList*. The meaning of every attribute is as follows:

placeName: the name of place, which represents a data's ID in composition code.

isALiveToken: the value of this attribute is True or False. True represents the place has one token, and False represents the place has no token.

tokenIsTo: parameter number ID; states which parameter token in this place will transfer to the service represented by post transition. For example, let a Web service S have three input parameters a, b and c, denoted as $S(a, b, c)$. Then, the parameter number of 'a', 'b' and 'c' is 0, 1 and 2, respectively.

inputList: The flow relation list from this place's preset to it. If the list is null, the place is an input place, and its variable value represented by the token in this place needs to be imported from outside.

outputList: The flow relation list from this place to its postset. If the list is null, the place is an output place, and its variable value represented by the token in this place needs to be exported to outside.

3.2.2 Transition

A transition represents a Web service, and has four attributes: *WSName*, *isALiveTransition*, *inputList*, *outputList*. The meaning of every attribute is stated as follows:

WSName: the name of service. According to this attribute, a corresponding Web service can be invoked in the process of creating executable code by Petri net model.

isALiveTransition: the state of this transition that denotes whether this transition can be fired or not.

inputList: the flow relation list from its preset to this transition.

outputList: the flow relation list from this transition to its postset.

3.2.3 Token

Token exists in place. Here we use the place's *isALiveToken* data item to represent token's presence.

3.2.4 Flow Relation

Flow relation describes dataflow of Web services composition. It has two attributes: *source* and *target*:

- source: the flow relation's start point, which may be a place or transition.
- target: the flow relation's end point, which may be a place or transition.

These two values of flow relation can be automatically filled according to the two points linked by this flow relation.

3.3 Petri Net Model of the Illustrating Example

For the illustrating example in Section 2, its Petri net model of the Web services composition is shown in Figure 2.

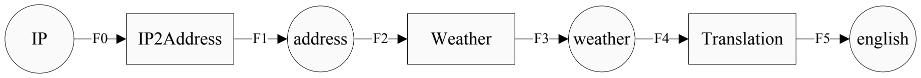


Figure 2. Petri net model of the illustrating example

The initial attribute values of places, transitions and flow relations are listed in Table 2, Table 3 and Table 4, respectively.

Attribute	**IP**	**address**	**weather**	**english**
placeName	IP	address	weather	english
isALiveToken	false	false	false	false
tokenIsTo	0	0	0	−1
inputList	Null	F1	F3	F5
OutputList	F0	F2	F4	Null

Table 2. Initial attribute values of places

Attribute	**IP2Address**	**Weather**	**Translation**
WSName	IP2Address	Weather	Translation
isALiveTransition	false	false	false
inputList	F0	F2	F4
outputList	F1	F3	F5

Table 3. Initial values of transitions' attributes

Attribute	**F0**	**F1**	**F2**	**F3**	**F4**	**F5**
source	IP	IP2Address	Address	Weather	weather	Translation
target	IP2Address	address	Weather	weather	Translation	english

Table 4. Initial values of flow relation's attributes

4 A COMPOSITION CODE CREATION ALGORITHM BASED ON PETRI NET MODEL

This paper designs an algorithm that can create executable Web services composition code automatically from Petri net model. Based on the Web services composition's Petri net model designed in Section 3.2, this method will induct users to create a Code file for saving the Web services composition code, then to run the painted Petri net based on Petri net's properties and to automatically import composition code of the Petri net model to the Code file in running process. The basic idea is shown below:

Step1. Initialize Petri net. If place p has no preset transition, put attribute *isALive-Token* of place p to True, write a parameter definition code to Code file.

Step2. Check all transitions. If all presets of *isALiveToken* attribute of transition t are True, change its *isALiveTransition* to True.

Step3. If t's *isALiveTransition* is True, fire transition t, and write code of invoking the Web service represented by transition t to Code file.

Step4. If there is a transition fired in Step3, turn to Step2, else turn to Step5.

Step5. Check the places that have no postset. If all of their token states are true, write the composite service's output code, and turn to Step 7. Else turn to Step6.

Step6. The logic of Petri net is wrong. The output state cannot be reached. Turn to Step7.

Step7. The run of Petri net is over.

Based on the Petri net model's data structure defined in Section 3.2, for a composite service's Petri net model $WSPN = (P, T, F)$, the algorithm for creating composition code in this paper is shown below:

Step1. Initiate Petri net, $\forall p \in P$, if ${}^{\bullet}p = \Phi$, then $p.isAliveToken :=$ True.
Write a parameter definition code to Code file; the code's basic form is shown below:
String placename = (data input from outside);
// The function of this code is to define a variable whose value should be got from outside interactively in the running process of the Web services composition code.

Step2. Check all transitions, $\forall t \in T$, if $\forall p \in^{\bullet} t$, and $p.isAliveToken =$ True, then $t.isAliveTransition :=$ True;

Step3. Check all transitions,
$\forall t \in T$, if $t.isAliveTransition =$ True, then Fire the transition t.
Moreover, for $\forall p \in t^{\bullet}, p.isAliveToken :=$ True; write following code of invoking the Web service to Code file;
String placename1 = wsname(String . . . placename);
// Where Placename is a list, placename1 is the value of the attribute *placeName* of t's postset, wsname is the value of attribute *WSName* of transition t. Placename is a list of *placeName* values of t's preset.

Step4. If there is a transition that has been fired in Step3, turn to Step2, else turn to Step5.

Step5. Check all places,
$\forall p \in P$, if $p^{\bullet} = \Phi$ and $p.isAliveToken =$ False, then the Petri net model is wrong; else, write the code of composite service's output to file below:
Print(placename);
where placename is value of attribute placename of place p satisfying parameter list of *placeName* property in place that satisfies $p^{\bullet} = \Phi$ and $p.isAliveToken =$ False and the token state of p is false.

Step6. Finish the Petri net run.

5 SERVICE COMPOSITION CODE AUTOMATIC CREATION TOOL

Web services composition code automatic generation tool is composed by three parts: Web service local pretreatment, Web service's Petri net model and the generation algorithm of executable code. Architecture of this tool is shown in Figure 3.

The localization pretreatment part is responsible for correspondence between local program and Web service, and creates localization JAR to help the automatic creation of executable code for local Web services composition. The Petri net model part uses Petri nets to simulate Web services, defines the graphical Petri net model, and paints the Web services composition logic visually with graphical Petri net model editor tool. The algorithm part automatically generates the executable Web services composition code according to the painted Petri net model in the second part.

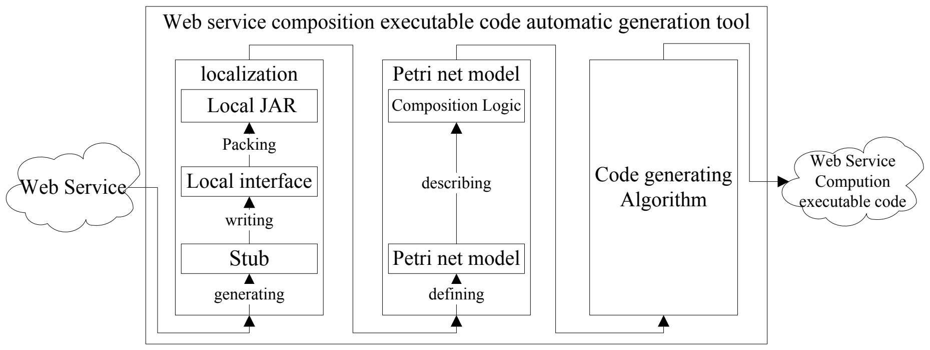


Figure 3. Architecture of the tool

Based on the tool system's structure description in Figure 3, the process of Web services composition and the process of executing the Web services composition code are shown in Figure 4.

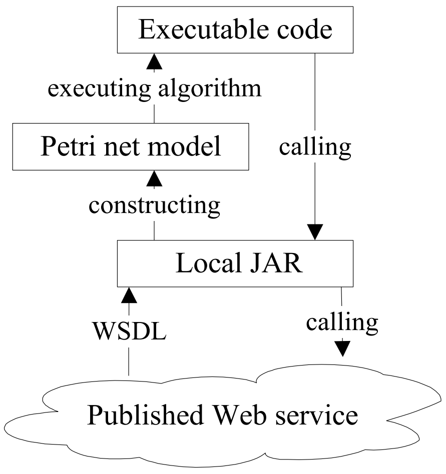


Figure 4. Process of executing the Web services composition code

5.1 The Localization Pretreatment of Web Service

In the process of developing Web services composition, we find that the type number and style in input and output messages are often different in a different Web service. Besides, the interface of Web service is described by XML, and if there is no local pretreatment before the communication between Web services, the developer should firstly deal with SOAP, WSDL and other protocols on their flat roof at every time of Web service invocation, and then should carefully distinguish and compile the parameters needed in the communications between Web services. However, those works are very fussy and error-prone, because the final composition code is very long and elusive, which makes it inconvenient to automatically generate Web services composition code from composition model. The steps of Web service localization pretreatment are as follows.

5.1.1 Create Stub

Use WSDL compiler to parse the specific Web service's WSDL document and automatically create the local stub, which will shield communication at the level of SOAP.

This paper uses Axis2 Code Generator as the WSDL compiler in the developing process of the tool. For a specific available Web service *IP2Address*, we can get the location of its WSDL document. The WSDL document location of *IP2Address* in this paper's example is: `http://Webservice.Webxml.com.cn/WebServices/IpAddressSearchWebService.asmx?wsdl`.

Take this location as the input of WSDL compiler Axis2 Code Generator, a stub IP2AddressStub.java will be generated to access the *IP2Address* service. This stub creates a correspondent inner class for every message and every operation, which is necessary for accessing the *IP2Address* service described in WSDL document. This paper's example will use the class of *GetCountryCityByIp* (transfer messages for service), *GetCountryCityByIpResponse* (return the result of invoking service) and so on.

After its creation, the stub can be used to invoke Web service. First, create an instance *IP2Address* for a class of stub IP2AddressStub. Second, create an instance *ip2addr* for a message inner class of *GetCountryCityByIp*, put the types data of IP address to the object *ip2addr*. Then, through instance *IP2Address*, *ip2addr* is used to invoke service *IP2Address*'s operation *GetCountryCityByIp*. Create an instance *response* to request inner class *IP2Address.GetCountrCityByIpResponse* again. At last, fulfil the invocation of service operation by *response* to invoke its own method.

5.1.2 Compile the Localization Interface

In the development process, we find that stub code automatically created by WSDL compiler is very complex. For example, the code for messages of *GetCountryCityByIp* has more than 600 lines in length, and the code for the invocation of *GetCountryCityByIpResponse* has more than 400 lines. If Web services composition is developed directly on such stub code, it is prone to go wrong and also very ineffective. On the other hand, the Web services used in the process of composition are generally developed by different companies and the naming styles for Web services and their operations are also different. Even more, there are some services with the same name. Last, in the process of invoking Web services, the messages from different operations have different types. All of these issues make the process of automatic creation from model to code very difficult.

For example, the data returned from *IP2Address* service listed in Section 2 is a string array with much information. For instance if the IP:123.165.173.66 is the input of *IP2Address* service, it will return data such as: "Telecom of Haerbin City, Heilongjiang Province", but the service *Weather* input can only receive a simple city name with fixed form, such as "Shanghai", "Haerbin" and so on. Therefore, it is needed to match data format of their input and output before composition for a group of given Web services on the Web.

For simplifying the model, we try not to depict the type information needed in invoking service on the model, so some pretreatments should be done for the generated stub before WSC modeling.

It can be known from the previous part that the work of invoking Web service from stub is finished by an instance of request class (Response). Therefore, for the developer who uses Web service through its stub, the only thing that s/he needs to know is how to invoke the request class (Response). In localization interface, an opResponse method is set up for every request class (Response) and s/he instantiates an object of that request class. With the Web services composition code, the invoking of a Web service can be finished only by invoking the method of opResponse and sending it to a parameter list.

5.1.3 Pack in JAR

Pack the result of A and B in JAR, and provide executable code to invoke. For the three given Web services as shown in Section 2 as follows:

- *IP2Address*: `http://Webservice.Webxml.com.cn/WebServices/IpAddressSearchWebService.asmx?op=getCountryCityByIp`
- *Weather*: `http://Webservice.Webxml.com.cn/WebServices/WeatherWS.asmx?op=getWeather`
- *Translation*: `http://fy.Webxml.com.cn/Webservices/EnglishChinese.asmx?op=Translator`

Based on Web service's WSDL location obtained directly from the domain name location of Web service, WSDL compiler Axis2 Code Generator can be used to create stub. Next, we use *Weather* service as an example to explain the implementation of localization interface.

According to the inputted IP, rich information will be returned by *IP2Address* service as its output, with a fixed form ***province***city. However, only a city name is required as the input of *Weather* service. So we can intercept the substring of the output of *IP2Address* to be the *Weather* service's input, such as the substring before the word of "city" and after the word of "province". Besides, if the *Weather* service cannot find the given city's weather, the original information returned is a null string. For this case, some additional information can be added in the localization interface to optimize user's experience; for example, if search is unsuccessful, it will return: "please input a right city name". The code is shown in Table 5 (some hard-to-understand code is ommited, such as exception handling).

```
public String getWeather(String address) {
    int first = 0, second = 0;
    first = address.indexOf(”province ”);
    second = address.indexOf(”city ”);
    if(first ¡ 0)first = 0;
    if(second ¡ 0)second = address.toCharArray().length;
    address = address.substring(first+1,second);//intercept a suitable substring
    WeatherWSStub.GetWeather getweat = new WeatherWSStub.GetWeather();
    getweat.setTheCityCode(address);//invoke service to get weather information
    WeatherWSStub.GetWeatherResponse weatResponse =
    weather.getWeather(getweat);
    if(weatResponse.localGetWeatherResult.localString.length ≥ 0)
    // if the search is successful, return the information got from Weather service
    return weatResponse.localGetWeatherResult.localString;
    //if the search is unsuccessful, return a suggestive information
    else return “please input a right local”;
}
```

Table 5. Localization code of Web Service Weather

5.2 Create Petri Net Model

The Web service Petri net modeling tool developed in this paper has two interfaces of graphical editor interface and code browse interface. Based on GEF framework, the graphical editor interface can edit graphical Petri net model and its attributes, and also has many assistant functions such as outline view, snap to grid, snap to center line, zoom, delete, undo, redo and other functions. The modelling process strictly observes the Petri net syntax. Flow relation can only exist between place and transition. In code browse interface, we can examine the Petri net code model that is synchronous with Petri net graphical model. It can also realize other functions such as copy code, and check the syntax's correctness of model and so on.

The graphical editor can be used to paint Web services composition's Petri net model graph as shown in Figure 5. In the model graph, the parameter represented by the token in place *ip* is inputted from outside (place *ip*'s preset is null). The parameter represented by token in place *result* is the output of the whole service composition (the place *result*'s postset is null). Place *address* is the *IP2Address*'s postset and also the *Weather*'s preset. There is shared data between service *Weather* and *Translation* too. That is, the place *weather* is transition *Weather*'s postset, and also preset of transition *Translate*.

After the visual Petri net model is painted, the daemon of the tool will automatically create the corresponding code model; the Petri net model of the example is shown in Figure 5.

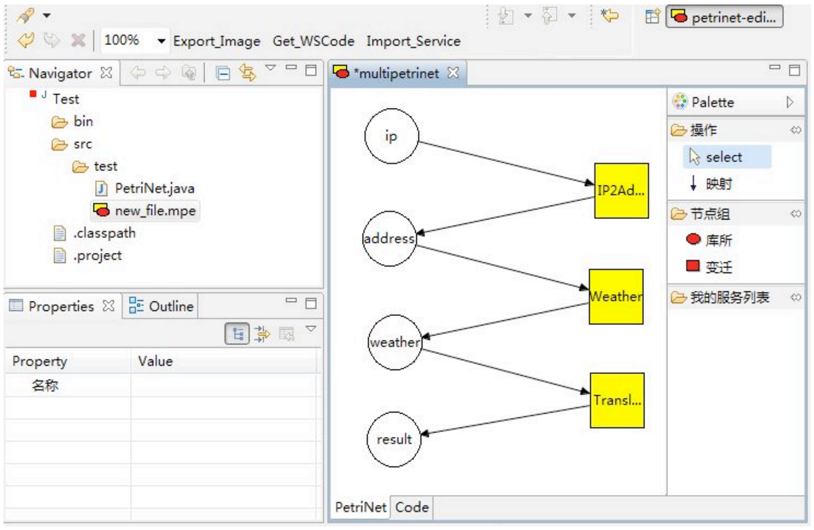


Figure 5. Petri net model interface of example

5.3 Create Executable Code

We can know from the Petri net graphical model of Figure 5 that place *ip* has no input transition, its parameters are from outside; place *result* has no output transition, its parameters are outputs of the whole service composition; and the parameters represented by the tokens in places *address* and *weather* are middle ones in running process of the whole service composition.

First, create a new Code file named EnglishWeather.java. Based on the algorithm in Section 4, the steps of creating executable code from the Petri net model in Figure 5 are shown below:

Step1. Initiate Petri net. Let the attribute *isALiveToken* of place *ip* be True, define a variable named *ip* in file EnglishWeather and initialize its value. See below: **String ip = JOptionPane.showInputDialog("", "please input ip");**

Step2. Scan all transitions. *isALiveToken* of place *ip*, which is transition *IP2Address's* preset, is True. Put the attribute *isALiveTransition* of *IP2Address* to True, and then put *isALiveToken* of place *ip* to False.

Step3. Scan all transitions. Because attribute *isALiveTransition* of transition *IP2Address* is True, fire the transition and input a section code to English-

Weather file for invoking service *IP2Address*. Search the postset place of transition *IP2Address*. Define a variable *address*, put the output of service *IP2Address* to variable *address*. Put attribute *isALiveTransition* of *IP2Address* to False and attribute *isALiveToken* of place *address* to True. The code format that inputs to EnglishWeather file in this step is shown below:
IP2Address ip2address1 = new IP2Address();
String address = ip2address1.ip2address(ip);

Step4. Repeat Step2 and Step3.
The value of attribute *isALiveToken* of place *address* (preset of transition *Weather*) is True, so put attribute *isALiveTransition* of transition *Weather* to True, and then put *isALiveToken* of place *address* to False. Fire transition *Weather*; input a section code to EnglishWeather file for invoking *Weather* service. Define a variable *weather* in EnglishWeather file and put the result of invoking *Weather* service to the variable. Put attribute *isALiveTransition* of transition *weather* to False and attribute *isALiveToken* of place *weather* to True. The code format that inputs to EnglishWeather file in this step is shown below:
Weather weather1 = new Weather();
String weather = weather1.weather(address);
The value of attribute *isALiveToken* of transition *Translation*'s preset place *Weather* is True, so put value of attribute *isALiveTransition* of *Translation* to True, and then put *isALiveToken* of place *weather* to False. Fire transition *Translation*; input a section code to EnglishWeather file for invoking *Translation* service. Define a variable *result* in EnglishWeather file and put the output of invoking *Translation* service to the variable. Put attribute *isALiveTransition* of transition *Translation* to False and attribute *isALiveToken* of place *result* to True. The code format that inputs to EnglishWeather file in this step is shown below:
Translation translation1 = new Translation();
String result = translation1.transition(weather);
Check all transitions, and there is no transition that can be fired.

Step5. Check the place whose postset is null. The value of attribute *isALiveToken* of place *result* is True. Write following code of composite service's output to EnglishWeather file:
System.out.println(result);

Step6. Finish the run of Petri net.

Based on the above steps, the executable service composition code created by the tool is shown in Table 6.

(Note: operater.* in line 2 of the code is the localization interface package of three Web services: *IP2Address*, *Weather* and *Translation*. The codes from the third to the fourteenth lines are generated by Petri net model. The codes from the fifteenth to the eighteenth lines are test codes for service composition.)

```
import javax.swing.JOptionPane;
   import operater.*;
public class EnglishWeather{
   public void compositionofWebservices(){
      String ip = JOptionPane.showInputDialog("", "please input ip");
      IP2Address ip2address1 = new IP2Address();
      String address = ip2address1.ip2address(a);
      Weather weather1 = new Weather();
      String weather = weather1.weather(b);
      Translation translation1 = new Translation();
   String result = translation1.translation(c);
   System.out.println(result);
   }
   public static void main(String[]args){
      EnglishWeather englishWeather = new EnglishWeather ();
   englishWeather.compositionofWebservices();
   }
}
```

Table 6. Executable service composition code (example)

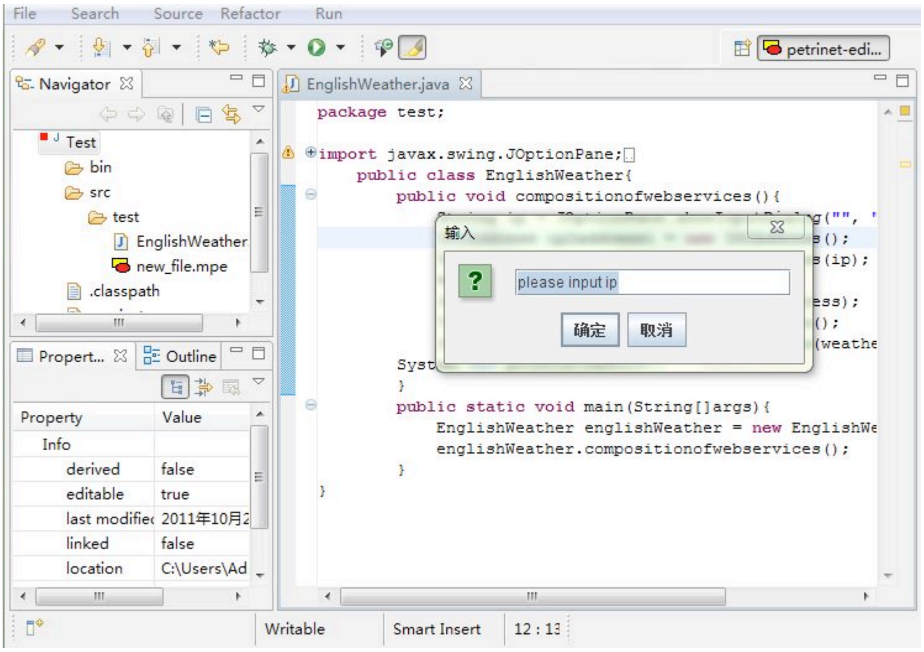
After running the composition code, user will be reminded to input IP address, see Figure 6.

Put the IP: "222.69.212.164" as an input; the composition code will invoke localization interface, and exchange with service *IP2Address* by stub. The result from *IP2Address* will be taken as the input of service *Weather*. Use the same process to invoke services *Weather* and *Translation*. At last, the composition code will return the English weather information at the location of JiaDing that IP "222.69.212.164" located. The resulting weather information "sunshine" is shown in Figure 7.

6 CONCLUSIONS

Web services composition is a way for effective utilization of Web services. For the Web services composition problem, this paper has designed a Web service's Petri net model and a graphical development tool, which can describe the services composition's logic structure in a convenient and shortcut way, and automatically create executable code from the model. Our further work includes two following aspects.

1. Follow the specification of PNML [12], the Petri Net model painted with the tool is output as a XML document, then some existing Petri Net analysis tools can be used, such as PIPE2 [13] for property analysis and verification, to ensure correctness of service composition's Petri net model. The work in this area has been basically completed, and the Petri net model produced in our tool can be converted into PIPE2 tool; then its properties such as reachability, boundedness and liveness can be analyzed and verified by PIPE2.



```
package test;

import javax.swing.JOptionPane;
    public class EnglishWeather{
        public void compositionofwebservices(){
                                                    g("", '
                                                    s();
                                                    s(ip);

                                                    ess);
                                                    ();
                                                    (weathe
        Syst
        }
        public static void main(String[]args){
            EnglishWeather englishWeather = new EnglishWe
            englishWeather.compositionofwebservices();
        }
}
```

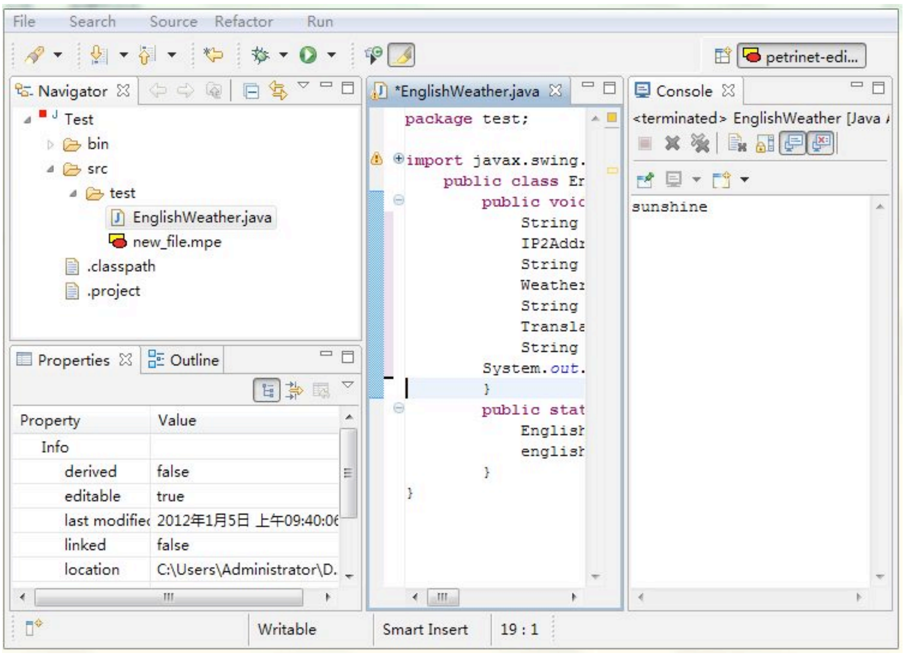
Property	Value
Info	
derived	false
editable	true
last modifie	2011年10月2
linked	false
location	C:\Users\Ad

Figure 6. Execution interface of created composition code

2. Current tools only support service composition processes of the sequence structure; for other structures such as select, parallel, and loop structures their executable service composition code automatic creation methods will be addressed in our further work.

Acknowledgments

This work is partially supported by National Natural Science Foundation of China under Grant No. 61173042 and 61105047, National Basic Research Program of P. R. China (973 Program) under Grant No. 2010CB328101, Program for New Century Excellent Talents in University under Grant No. NCET-10-0598, "Shu Guang" Project supported by Shanghai Municipal Education Commission and Shanghai Education Development Foundation under Grant No. 10SG23 and Shanghai Rising-Star Program under Grant No. 12QH1402300, HongKong, Macao and Taiwan Science and Technology Cooperation Program of China No. 2013DFM10100.



```
package test;

import javax.swing.
    public class Er
        public voic
            String
            IP2Addr
            String
            Weather
            String
            Transla
            String
        System.out.
        }
        public stat
            English
            english
        }
}
```

```
<terminated> EnglishWeather [Java
sunshine
```

Property	Value
Info	
derived	false
editable	true
last modifiec	2012年1月5日 上午09:40:06
linked	false
location	C:\Users\Administrator\D.

Figure 7. Execution interface of created composition code

REFERENCES

[1] Bartalos, P.—Bieliková, M.: Automatic Dynamic Web Service Composition: A Survey and Problem Formalization, Computing and Informatics, Vol. 30, 2011, No. 4, pp. 793–827.

[2] Martin, K.—Jonathan, C.—Sarah, H. et al.: BPEL4WS Business Processes with WebSphere Business Integration: Understanding, Modeling, Migrating. IBM Redbook, 2004.

[3] Oracle BPEL Process Manager Overview, Availaible on: `http://www.oracle.com/technetwork/middleware/bpel/overview/index.html`.

[4] Protégé OWL ontology Editor, Availaible on: `http://protege.stanford.edu/plugins/owl/documentation.html`.

[5] Thomas, J. P.—Thomas, M.—Ghinea, G: Modeling of Web Services Flow. In: Proceedings of the 2003 IEEE International Conference on E-Commerce Technology (CEC '03), 2003, pp. 391–399.

[6] Schlingloff, H.—Martens, A.—Schmidt, K.: Modeling and Model Checking Web Services. Electronic Notes in Theoretical Computer Science, Vol. 126, 2005, pp. 3–26.

[7] Narayanan, S.—McIlraith, S.: Simulation, Verification and Automated Composition of Web Services. In: Proceedings of the Eleventh International World Wide Web Conference (WWW-11), 2002, pp. 77–88.

[8] Ding, Z. J.—Wang, J. L.—Jiang, C. J.: An Approach for Synthesis Petri Nets for Modeling and Verifying Composite Web Service. Journal of Information Science and Engineering, Vol. 24, 2008, No. 55, pp. 1309–1328.

[9] Web Service, Availaible on: `http://Webservice.Webxml.com.cn`.

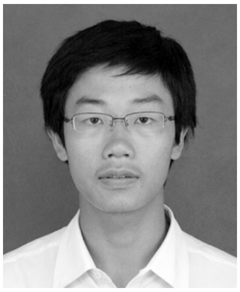
[10] Ding, Z. J.—Liu, J. Q.—Wang, J. L.: A Petri Net Based Automatic Executable Code Generation Method for Web Service Composition. In: Proceedings of the 2012 International Conference on Information Technology and Software Engineering (ITSE 2012), Lecture Notes in Electrical Engineering, Vol. 212, 2013, pp. 39–48.

[11] Murata, T.: Petri Nets: Properties, Analysis and Applications, In Proc. of the IEEE, Vol. 77, 1989, No. 4, pp. 541–580.

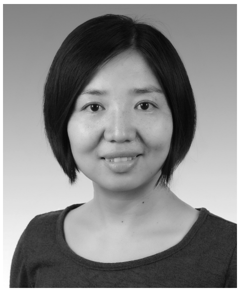
[12] PNML: the Petri Net Markup Language, Availaible on: `http://www.pnml.org/`.

[13] PIPE2: Platform Independent Petri net Editor 2, Availaible on: `http://pipe2.sourceforge.net/`.

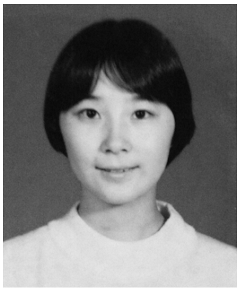
Zhijun Ding received the M. Sc. degree from Shandong University of Science and Technology, Taian, China, in 2001 and the Ph. D. degree in computer science from Tongji University, Shanghai, China, in 2007. He is currently an Associate Professor at the Department of Computer Science and Technology, Tongji University. He has published more than 60 papers in domestic and international academic publications. His research interests are in service computing, semantic Web, formal engineering, Petri nets, and workflows.



Jieqi Liu received the B. Sc. degrees from Xiangtan University, Xiangtan, China, in 2009. He is currently working toward the M. Sc. degree in the Department of Computer Science and Technology, Tongji University. His research interests include service composition, Web services, and Petri nets.



Junli Wang received the Ph. D. degree in computer science from Tongji University, in 2007. She is currently an Associate Research at the College of Electronics and Information Engineering, Tongji University. Her research interests are in service computing, semantic Web, ontology learning.



Fang Wang is a lecturer in the Department of Information Systems and Computing of Brunel University. She received Ph. D. in artificial intelligence from the University of Edinburgh. She worked as senior researcher in the research centre of British Telecom Group, before she joined Brunel University in 2010. Her research interests include software agents, cognitive neuroscience and distributed computing. She has published many papers in books, journals and conferences, filed a number of patents and received several technical awards.