Intelligent Web Based on Mathematic Theory

Case Study: Service Composition Validation via Distributed Compiler and Graph Theory

Ahmad Karawash, Hamid Mcheick, and Mohamed Dbouk

Abstract. This paper discusses a model for verifying service composition by building a distributed semi-compiler of service process. In this talk, we introduce a technique that solves the service composition problems such as infinite loops,deadlock and replicate use of the service. Specifically, the client needs to build a composite service by invoking other services but without knowing the exact design of these loosely coupled services. The proposed Distributed Global Service Compiler, by this article, results dynamically from the business process of each service. As a normal compiler cannot detect loops, we apply a graph theory algorithm, a Depth First Search, on the deduced result taken from business process files.

Keywords: SOA (Service Oriented Architecture), Compiler, Business Process Execution Language (BPEL), Depth First Search (DFS), Distributed Global Service Compiler (DGSC).

1 Introduction

Web services are defined as self-contained, modular units of application logic which provide business functionality to other applications via an Internet connection. Web services support the interaction of business partners and their processes

Ahmad Karawash · Hamid Mcheick
Department of Computer Science, University of Quebec at Chicoutimi (UQAC),
555 Boulevard de l'Université Chicoutimi, G7H2B1, Canada
e-mail: {ahmad.karawash1,hamid_mcheick}@uqac.ca

Mohamed Dbouk
Department of Computer Science, Faculty of Sciences (I), Lebanese University,
Rafic-Hariri Campus, Hadath-Beirut, Lebanon
e-mail: mdbouk@ul.edu.lb

by providing a stateless model of atomic synchronous or asynchronous message exchanges (B. Srivastava and J. Koehler). Every service consists of a domain of computers and computers in a domain can just communicate with each other through predefined functions (Amirjavid F. et al., 2011). Services can be invoked by other services or applications. Services are designed for interaction in a loosely coupled environment, and therefore are an ideal choice for companies seeking inter or intra business interactions that span heterogeneous platforms and systems (K. LI, 2005).

In many cases, a single service is not sufficient to the user's request and often services should be combined through service composition to achieve a specific goal. Nowadays, researches show that problems of composing web services are expressed in designing, discovery, validation and optimization of service composition. The paper proposes a way of dynamic validation of service composition to prevent some errors (infinite loops, blocked services, incorrect business flow design and many others) at the beginning of the design phase of new composite service.

Before designing a new composite service, the service discovery process returns a set of candidate services and at some of those services are used in the composition process according to non-functional criteria (cost, time, and context-aware). But nothing, in this discovery notifies service developer if the invoked services are working normally or not.

In the dynamic world of service-oriented architectures, however, what is sure at design time, unluckily, may not be true at run time. The actual services, to which the workflow is bound may change dynamically perhaps in an unexpected way, may cause the implemented composition to deviate from the assumptions made at design time. Traditional approaches, which limit validation to being a design time activity, are no longer valid in this dynamic setting. Besides performing design-time validation, it is also necessary to perform continuous run-time validation to ensure that the required properties are maintained by the operating system. The compiler is the only way to validate the sequence of service process. It is the program that translates one language to another. Thus our goal is to implement a distributed dynamic compiler that compiles the composition of every new composite service. When a client designs a new composite service, the related compiler Grammar rules, of the invoked services, are sent to him as XML files then combined together to constitutes a local compiler that validate new service composition at design phase.

Section 2 describes the previous methods of service validation. Section 3 gives two service composition examples to highlight the problem of service validation. While section 4 proposes a new service composition validation model in the service design development phase; two techniques of parsing and depth first search are described and a simulation steps are given in this section. Section 5 summarizes the ideas as a conclusion and gives perspectives for future works.

2 Background

This part describes the previous works and some basic features.

2.1 Related Works

Some ways of dynamic system validation are discussed in this section. In 2006 Colombo et al., undertake the topic of dynamic composition where the service parts do not always behave along expected lines. They provide an extension to the BPEL language in the form of the 'SCENE platform' which addresses this issue. The proposed platform is validated forming an application using a set of real services and observing the behavior of the application (Colombo et al, 2006). In 2009, Silva et al proposed the DynamiCos structure which response the requirements of different customers to dynamically put together personalized services. To confirm the proposed structure they set together an extensive model of the structure which enables services to be deployed and be published in a UDDI-like registry (Silva et al, 2009).

In 2008, Eid et al explain a set of scales alongside which to evaluate the various frameworks of dynamic composition. The set of scales is inclusive and is roughly classified into three parts: input subsystem, composition subsystem, and execution subsystem. To be considered good a composition model must achieve well against these scales (Eid *et al.*, 2008).

In 2007, Shen et al found the Role and Coordinator (WSRC) model to hold dynamism in web service compositions. In this model, the development of service composition is divided into three layers: Service, Role, and Coordinator. To validate the model, the authors describe a case-study of a vehicle navigation system which comprises a global positioning system and a traffic control service (Shen *et al.*, 2007).

These are a small list of the validation methods in use for dynamic composition models and structures. Some of these methods are quite complex like the model proposed by DynamiCos or that of the SCENE platform. While, the validation ways that are simple like the case-study in the WSRC model seem useless.

The validation model proposed in this paper, Global compiler service, relatively simple as it expands the validation in one operating system to achieve a wide convention between all the operating systems that deal with service composition.

2.2 Basic Features

Business process execution language (BPEL) - BPEL is a language created to compose, orchestrate and coordinate web services. It is the result of over ten years of collaborative effort in Business Project Management by Microsoft and IBM. It provides both synchronous and asynchronous interactions and it gives suitable forms to create a process. It allows the creation of composite processes with all its related activities. BPMN steps: invoking web services, waiting for clients to invoke the service via message, generate response, manipulate variables, throws exceptions, pause for a selected time, terminate.

Compiler - is a program that takes a source program typically written in a high-level language and produces an equivalent target program in assembly or machine language (Aho I. et al, 2007). Also it is reports error messages as a part of the translation process. A compiler performs two major tasks: analysis of the source program and synthesis of the target-language instructions. In order to build a compiler, there are six phases to follow as in figure 1: i) scanning the input program will be grouped into tokens, ii) parsing or syntax analysis, iii) building a Context-Free-Grammar, iv) applying semantic analysis to keep on mapping between each identifier of data structure (symbol table) and all its information and ensure consistent, v) extracting assembly code generation and vi) finally realizing code optimization.

Depth First Traversal (DFS) – it is a graph theory algorithm for traversing a graph. It is a generalization of preorder traversal. It starts from a vertex and recursively it build a spanning forest that determine if the graph is cyclic (contain cycle loop) or acyclic.

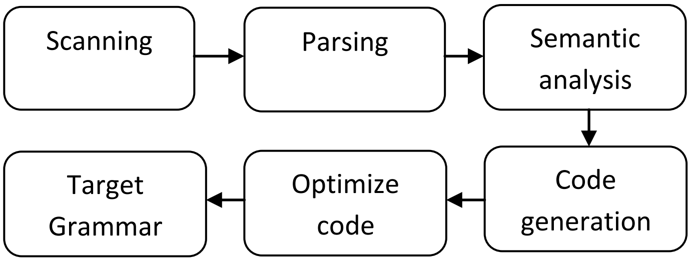


Fig. 1 Phases to build a compiler

3 Composition of Web Service

In order to highlight on the problem of web service composition and simplify the idea for the reader, this section gives two examples about service composition. The first example reflects a simple normal composition while the second shows an abnormal service behavior.

3.1 Simple Services Composition Example

Figure 2 shows a simple example of how Providers of web services are communicated to achieve a composed service.

Let *Client*$_2$ has to solve two mathematical formulas: "***F1: A = 2*x +3*y***" & "***F2: B = 2*x*** "

In order to achieve his goal *Client*$_2$ will design a new composite web service. First of all, he searches in *UDDI*$_2$ which gives him a summary about the services that are existed in the *Provider*$_2$. *UDDI*$_2$ has two services that solve two equations: *EQ*$_1$:"***2*x***" & *EQ*$_2$: "***3*y***"

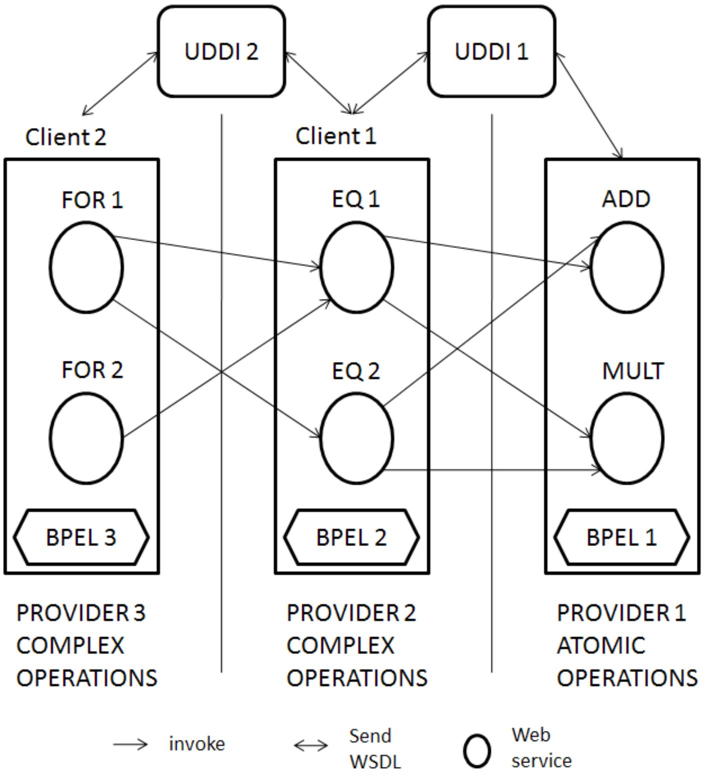


Fig. 2 Example of composite services

Using the information given in the *WSDL* file by *UDDI*$_2$, *Client*$_2$ invokes *Provider*$_2$ operations. But the two services *EQ*$_1$ and *EQ*$_2$ invoke other services *ADD & Multiply* from the *Provider*$_1$ to complete the required answer. This is a simple idea about how service composition works.

3.2 *Infinite Loop Example*

Web services are distributed through the whole internet and controlled by various sides. In the modern state, services are dynamically managed. Because the most used services are big and composite, the states of failure and infinite loop are detected sometimes. Failure of composite services results from an obstacle in one of its parts, while infinite loop exists as a result of wrong process flow design.

A real example of the service composition problem (infinite loop) is the TIBCO web service. See the link below (https://www.tibcommunity.com/message/70086). Figure 3 shows an infinite loop (or cycle) while executing composite service.

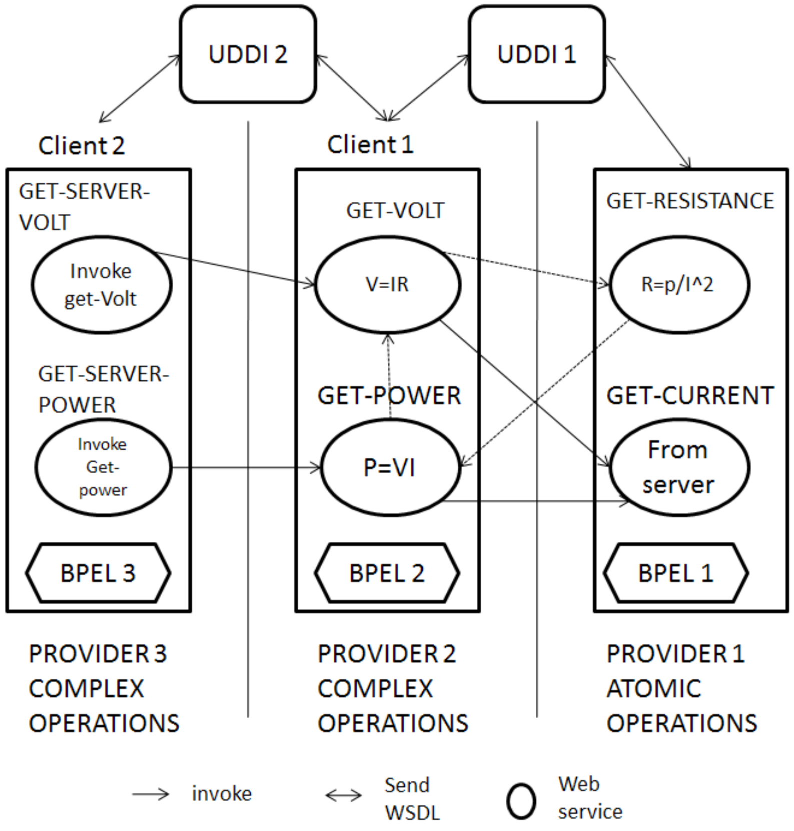


Fig. 3 Infinite loop of web service

Let voltage represented by *V*, current by *I*, resistance by R and represent power by *P*.

We have a set of service to use:

- *Client₂* build two services ***GET-SERVER-VOLT*** & ***GET-SERVER-POWER***
- *Provider₂* provides two services ***Get-Volt*** (***V= I*R***) & ***Get-Power*** (***P=V*I***)
- *Provider₁* provides two services ***Get-Resistance & Get-Current*** (***R=p/I^2***)

Client₂ wants to calculate the consumption of **Voltage and Power** of the last service provider machine during a composite service process. To complete the needed service, *Client*$_2$ invokes services from *Provider*$_2$ while *Provider*$_2$ invokes other services of *Provider*$_1$ to answer the question of *Client*$_2$. To build his own services (***GET-SERVER-VOLT*** & ***GET-SERVER-POWER***), *Client*$_2$ firstly searches in UDDI$_2$ about services and invokes **G*et-Volt*** & ***Get-Power*** from *Provider*$_2$.

Regarding the service "***Get-Volt***", it invokes *Provider*$_1$ (the last service provider in this process) services specifically the "***Get-Resistance***" service to calculate resistance '*R*' and it invokes the "**Get-Current**" service to calculate current '*I*'.

From the other side, the service "***Get-Resistance***" invokes "***Get-Power***" service from *Provider*$_2$ in order to calculate power '*P*'. But the service "***Get-Power***" invokes "***Get-Volt***" service to calculate voltage V.

Indeed, the "***Get-Server-Volt***" service falls into an ***infinite loop*** as seen above (figure 3) in red color. The "***Get-Volt***" node invokes the "***Get-Resistance***" node which needs results from "***Get-Volt***". Thus "***Get-Volt***" invokes itself indirectly. There are also other types of errors may occur because of partial fail or bad service communications.

4 Distributed Global Service Compiler (DGSC)

Our DGSC model consists of extracting compiler Context-Free-Grammar rules of the business process (BPEL) of a web service (figure 4). Then save these rules in the UDDI registry. Grammar rules are used later by the client when he fetches the registry to build a new composite service.

DGSC is simply a verification of new composite service before the execution that is valuable. Beside the development of programming languages, there exist many tools today, for example ATLAS in Eclipse, that transforms the design phase of service to related code automatically. But how achieve the same for online dynamic service composition without dangerous errors? As we know, it is impossible to discover the business process for web service through SOA model (meta-data about service process is just published by WSDL file).

Our goal is to discover design errors in the design phase of composite service without knowing the exact flow of service process. In fact, there are many obstacles facing our DGSC because the service design takes place in the client side and the content of web services is dynamically edited from several sides.

Service designer (Client side) searches the UDDI in order to build a new composite service. But nothing verifies that the new combination of service, that may also invoke other services, is free of errors and infinite loops. Also even if a correct composition of a complex service is achieved, this action may fail later because services are dynamically edited.

The proposed solution uses two phases of compiler design (scanning and parsing). This phase of the compiler is applied in the business process (BPEL) of service that contains the internal service design. A grammar rules drive similar to the case of the third phase of compiler design (Context-Free-Grammar). These rules are sent to UDDI registry in XML format. But the client uses the WSDL files of several services to design a new composite service. Thus depending on our

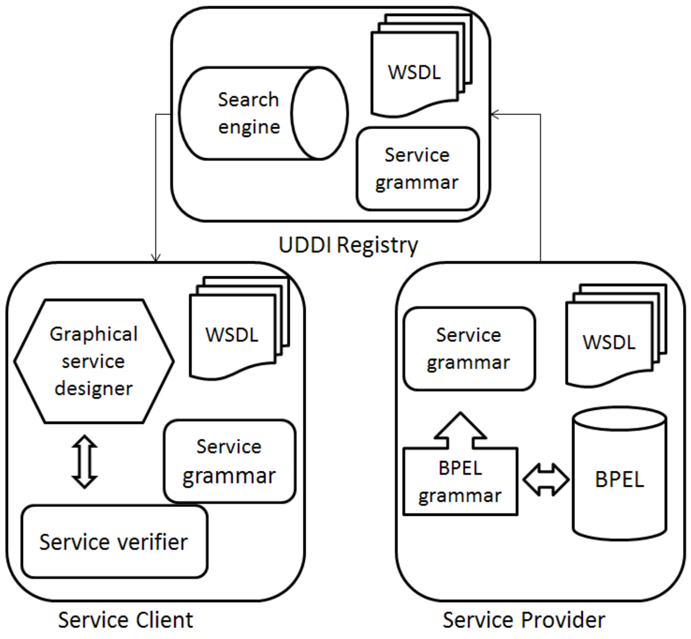


Fig. 4 Distributed global Service Compiler (DGSC)

proposed model he downloads the rules file, from the UDDI, with the WSDL file and he uses these rules to compile a new design of composite service. Locally on the client side, a mathematical algorithm (DFS) is applied in these rules to detect if the new design of composite service contains infinite loop before service deployment.

4.1 Extraction of Service Grammar

For every programming language there exists a compiler Grammar that is used to verify the steps of building a new program. As mentioned in section 2.2, there are six phases to build a compiler. But in our distributed compiler model we just need to detect infinite loops. Thus scanning then parsing is applied to the BPEL file and a result is a context-Free-Grammar for the service (similar to the third phase of building compiler).

To achieve our BPEL parser, we used the BPEL grammar of BPLE4WS written by the members of the ReDCAFD Laboratory at the University of Sfax.

The BPEL parser is implemented using Netbeans 6.9.1 and Java code. The outcome of the parser is a database entry in a specific table.

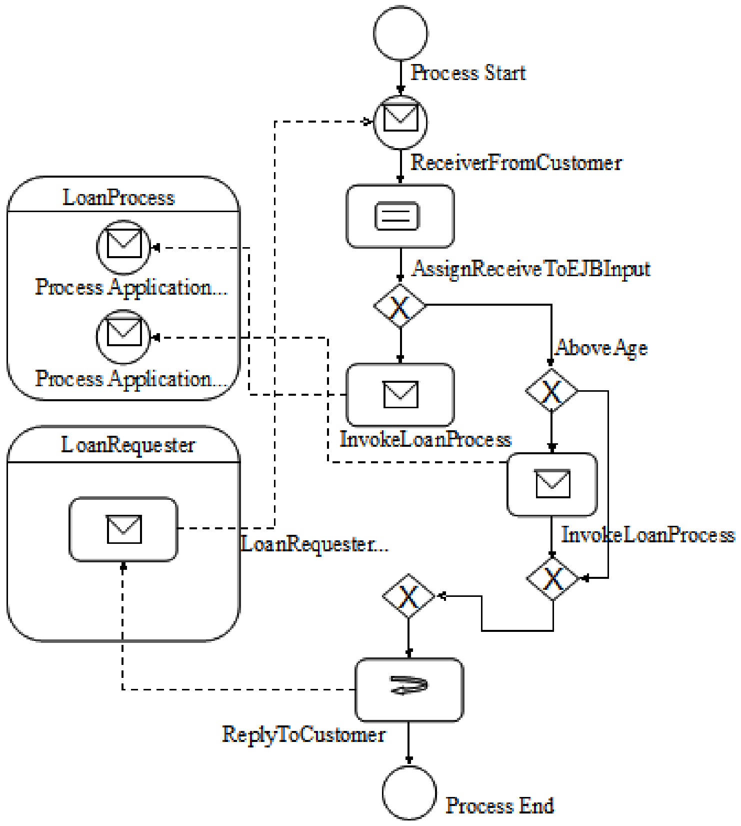


Fig. 5 Loan BPEL example

Table 1 The Output result of parsing the Loan BPEL code

Activity name	0	1	2	3	4	5
Activity Name	receive	if	invoke	if	invoke	Reply
Current State	ReceiveFrom-Customer	nul l	invokeloan-Process	nul l	invokeloan-Process	ReplyToCus-tomer
Conditions	LoanRequesto-rOpera.	nul l	ProcessAppli-cation	nul l	ProcessAppli-cation	LoanProces-sor
Partner Link	LoadRequestor	nul l	LoanProcessor	nul l	LoanProcessor	LoanProces-sor
Opera-tions	LoanRequesto-rOpera.	nul l	ProcessAppli-cation	nul l	ProcessAppli-cation	LoanProces-sor
Next Activity	1	2,3	5	4,5	5	6

Each row entry represents the details of an individual activity which provides information about the current state name, current state properties (as My Role, Partner Role), PartnerLink (which represents the associated web service), name of the operation being invoked, condition of a looping structural activity, current state number, and next possible state numbers. The result of parsing BPEL file is saved in an Excel file. Table 6 below contains the output of parsing Loan BPEL file in and the BPEL design is found in figure 5.

4.2 Detect Cycle by DFS

According to our model and after he requests the WSDL file from UDDI, the Compiler rules in XML format will be received by the client. In this section, the results of parsing the BPEL file will be considered as an inputs and these inputs will be transformed into a direct graph (arcs between nodes have sense).

Now the problem is changed from programming into a graph theory problem (figure 6).Instead of checking if the new design of composite service falls in infinite loop or not, we can verify if that the obtained graph is directed cyclic or acyclic. DFS algorithm is used to detect if the graph is acyclic.

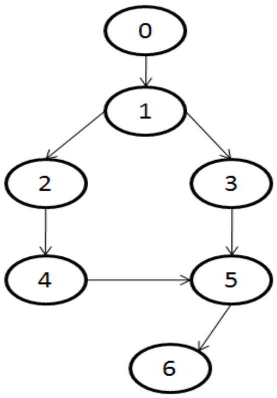


Fig. 6 The directed graph of the BPEL example

Indeed, DFS starts from the root node and explores siblings as far as possible along each branch before backtracking and if it arrive a visited node again it will notify that the graph is cyclic (it contains cycle). But sometimes the service designer need to have a cycle like while-loops, for-loops or even reply to node that sends a request. Thus in all cases we give the designer the permission to discard the detected harmful loops.

4.3 Distributed Compiler Concept

Composite services are built by invoking other already implemented services. But the web services are dynamic and able to be edited at random time.

In our case, the compiler is decentralized (federated) and the Grammar of every service works as a small compiler. When service designer forms a new composite service and the Grammar rules of all what is needed to be invoked services are collected at the client's machine. These rules are combined and DFS is applied to detect errors (infinite loops, errors...etc.). If the result returns an error then a notification appears to the web service designer that he must change the wrong graphical service design and the compiler gives him details about the error.

DGSC deals with the existing implemented services as standards, which have correct design, for new composite service.

In other words, if a developer wants to develop new composite service called ***XY*** and he needs to use other services then all these other services will stay non-editable at the last stage of designing this service. Thus the developer will receive an update message if the service he needs to invoke change during his design of new composite service. There several scenarios may occur during the design phase of new composite service:

- The relations between the services to be invoked are edited.
- One of the services to be invoked is being deleted or failed.
- An internal change occurs in the behavior of one of the services to be invoked.

Thus the server sends updates to a web service designer about any change occur in the required services of the new composition. Also the server prevents changes in these needed services while the deployment phase of the new composite service takes place. After the deployment of this new service it will be standard for other new services during the design phase.

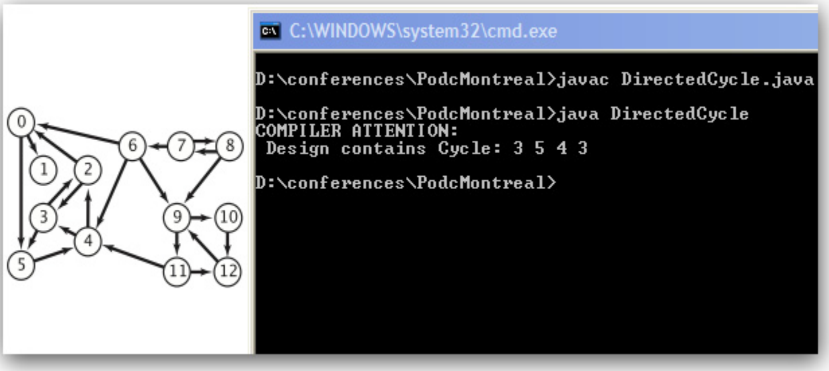
4.4 Steps of DGSC Simulation

In this simulation we have used Netbeans6.9.1, Apache Tomcat server, Apache-ode-war-1.3.5. We use http://ode.apache.org to execute a BPEL file based on the Apache server.

There are several steps in our simulation (check the sections 4.1 for detail of this simulation):

1. Start the Apache server and Netbeans.
2. Create database then a table (we used EXCEL file) to save the result of parsing a BPEL.
3. Apply the database connection between Netbeans and the database.
4. Use existing or Design some BPEL processes examples (as LoanProcess in our case) in service composition
5. Execute the code of our parser and put the BPEL file which previously designed as input and choose the excel database to save the output.
6. The XWTransformer class transforms the output result in the table to XML format and inserts its content to the WSDL file of the service.

7. Apply SOAP to send the WSDL content file from server to client (for more details refer to http://ode.apache.org/war-deployment.html).
8. Execute WXExctractor class code that extracts the parsing result from the WSDL file.
9. Combine the parsing results of all the services invoked in this composition.
10. Execute the Depth first search (DFS) code using the combined result as input.



```
D:\conferences\PodcMontreal>javac DirectedCycle.java

D:\conferences\PodcMontreal>java DirectedCycle
COMPILER ATTENTION:
 Design contains Cycle: 3 5 4 3

D:\conferences\PodcMontreal>
```

Fig. 7 The result DGSC simulation

This simulation validates the composition of new services. Before deploying the new design composite service, if the simulation detects infinite loop, it will notice the developer to fix it and then change the design. This model helps designer to avoid the cyclic composition service in the real world where a huge number of invocations take place and all the design services are dynamically changed and composed. Figure 7 shows the result of the DGSC simulation that is applied on a group of BPEL files of the services to be invoked to form a new composed service.

5 Conclusion and Future Works

The web service revolution simplified the building of new complex services because it gives the facility to invoke any type of services. On the other side, this revolution offers only the meta-data to the user about the service to be invoked. Thus invoking blindly another service may not give always the needed result. From this point, we start searching about a way to validate the web service at the design phase and the answer is the Distributed global Service Compiler. Since we have millions of services we cannot build a centralized compiler for all the services, we have developed a dynamic compiler per each new composite service. To deal with business process of the service, we started from BPEL and extract the Grammar rules, which represent the internal map (sequence) of that service. As the client contacts the Registry of services to get summary (such as WSDL file) before building a service, DGSC proposes to insert the Grammar rules in the WSDL files of the registry. These rules will be downloaded later by the user (when he invokes WSDL file) at the beginning of the design phase. The DGSC model helps us to validate sequence of the services.

As a future work, our goal is to achieve an intelligent organized web depending on the theorems of Mathematic. Taking this article as a case study, we reach a logical composition of service that can change its job according to client response without destroy the behavior of other services.

References

Srivastava, B., Koehler, J.: Web Service Composition - Current Solutions and Open Problems. In: IBM India & Switzerland Research, ICAPS 2003 Workshop on Planning for Web Services, vol. 35 (2003)

Amirjavid, F., Mcheick, H., Dbouk, M.: Job division in service oriented computing based on time aspect. Int. J. Communication Networks and Distributed Systems 6(1) (2011)

Li, K.: LUMINA: Using WSDL-S For Web Service Discovery. Master Thesis; University of Georgia (December 2005)

Colombo, M., Di Nitto, E., Mauri, M.: SCENE: A service composition execution environment supporting dynamic changes disciplined through rules. In: Dan, A., Lamersdorf, W. (eds.) ICSOC 2006. LNCS, vol. 4294, pp. 191–202. Springer, Heidelberg (2006)

Silva, E., Pires, L.F., van Sinderen, M.: Supporting dynamic service composition at runtime based on end-user requirements. Centre for Telematics and Information Technology University of Twente, The Netherlands P.O. Box 217, 7500 AE Enschede (2009)

Eid, M.A., Alamri, A., El-Saddik, A.: A reference model for dynamic web service composition systems. International Journal of Web and Grid Services (2008)

Shen, L., Li, L., Ren, S., Mu, Y.: Dynamic composition of web service based on coordination model. In: The Joint International Conferences on Asia-Pacific, Web Conference and Web-Age Information Management (2007)

Aho, A.V., et al.: Compilers, principles, techniques, and tools, QA76.76.C65A37 (2007)