

WS-CDL+: An Extended WS-CDL Execution Engine for Web Service Collaboration

Zuling Kang, Hongbing Wang
School of Computer Science and Engineering
Southeast University
zelkey@msn.com, hbw@seu.edu.cn

Patrick C.K. Hung
Faculty of Business and Information Technology
University of Ontario Institute of Technology
Patrick.hung@uoit.ca

Abstract

Web services are becoming the prominent paradigm for distributing, computing, and electronic business, while there is an increasing surge to provide online Business-to-Business collaborations. The Web Services Choreography Description Language (WS-CDL) is a Web service specification developed by W3C, in order to provide peer-to-peer collaborations for participants from different parties. Despite the great research interests it has received during recent years, no practical or even prototype execution engine has been built for WS-CDL, which is, however, essential to test and evaluate the properties of WS-CDL when doing research on it, and promote its application fields in business. This paper implements an execution engine of WS-CDL, which has never been built before, and experiments on the functionalities and performance of the engine. We also address the extensions toward WS-CDL, namely WS-CDL+, which are built into our execution engine. Finally, the whole paper is concluded, addressing the application perspectives of WS-CDL/WS-CDL+.

1. Introduction

The promise of reusability in software systems has become a common theme in commercial application development in recent years. Today we see it with Web services and the generic concept of Service-Oriented Computing, namely SOC [11]. Web services are designed to support interoperable machine-to-machine interaction over a network, and we use it to implement the SOC-based development using the composition technologies, which define an executable process to be enacted by a central orchestration engine, and the collaboration technologies, which describe an interaction protocol to be enacted by the peers without any central entity governing the collaboration. WS-CDL [1][2][3][4] is one of the most promising collaboration technologies.

Our work aims to advance the current state of the Web service collaboration technology by addressing the following two issues: **a) the prototype implementation**

of a WS-CDL+ execution engine. The study of WS-CDL calls for an execution engine; so that we can experiment on the properties we are working with, and develop applications using WS-CDL. However, today's research mainly focuses on the description, translation and semantic of WS-CDL, leaving no execution engines being built. Implementing a WS-CDL execution engine is quite helpful to the research of WS-CDL itself, as well as its communication and coordination protocols/models. The availability of a WS-CDL execution engine is also crucial to bring WS-CDL into application field, enabling a WS-CDL document to *run* on servers. Moreover, with the prototype system, researchers and developers are able to simulate their choreography process and get the results of choreography in a single computer without having to interact with the real WS-CDL/WS-CDL+ participants, which makes testing and debugging WS-CDL/WS-CDL+ documents easy. And **b) the extension of the WS-CDL specification into WS-CDL+.** It is no debut that the idea underlying WS-CDL is fascinating; however, when it comes to the application field, its expressiveness and usability become questionable. To enhance it in this way, we introduce six extended functionalities into the WS-CDL+ execution engine, including the user-defined functions, the interaction model extensions, the implicit finalization mechanism, and etc.

It is worth noting that the communication model of WS-CDL+ can also be applied to other distributed computing systems, not restricted to WS-CDL+. Since the communication model is designed to adapt to applications that are built on various underlying communication protocols and work in a peer-to-peer mode, any model that is based on multi-protocols, and works in a peer-to-peer mode, can make use of this communication model.

The reminder of the paper is organized as follows: Section 2 provides an overview of the WS-CDL/WS-CDL+. Section 3 discusses how WS-CDL+ extends WS-CDL by addressing six extension mechanism. Section 4 illustrates the implementation of the execution engine, covering the main implementing details including the overall architecture, the communication model, the coordination functionality, etc. Section 5 tests the

execution engine, including its features and performance. Section 6 discusses the current state of the art in the Web composition/collaboration technologies. Section 7 presents the application perspective of WS-CDL/WS-CDL+, and concludes the paper.

2. Preliminaries

WS-CDL [1][2][3][4] is an XML-based language that can be used to describe the common and collaborative, observable behavior of multiple services that need to interact in order to achieve certain goals [3]. It describes this behavior from a global or neutral perspective rather than from the perspective of any one party, which is a key difference from that of BPEL4WS [19].

When we were building and testing the execution engine, we found it necessary to add extended features to the current WS-CDL specification, which finally leads to the formation of WS-CDL+. WS-CDL+ adds six new functionalities such as user-defined functions, timers, implicit finalization mechanism and so on, and imposes some restrictions to what WS-CDL has not explicitly stated, which increases the co-operability and reduces the arbitrariness between different WS-CDL/WS-CDL+ implementations as well.

The program model of WS-CDL+ is built upon that of WS-CDL [3]. The following presents that model, with the same contents of WS-CDL omitted:

- ✧ roleType, relationshipType and participantType. The same as that of WS-CDL.
- ✧ informationType, variable and token. With the variable initialization mechanism added.
- ✧ constant. With the environment variable feature added.
- ✧ function. Used to implement user-defined function.
- ✧ choreography. With the runIf and runWhile properties added to support the timer mechanism, and the finalizeMechanism property to decide whether to use the implicate finalization mechanism.
- ✧ channelType. The same as that of WS-CDL.
- ✧ workunit. The same as that of WS-CDL.
- ✧ activities and ordering structures. Adding three new basic activities, <output>, <assess> and <raiseException>, to support the debug and exception functionality.
- ✧ interaction activity. Similar with that of WS-CDL.
- ✧ semantics. The same as that of WS-CDL.

The WS-CDL+ execution engine is built using the Java programming language, and executing on top of the BPEL4WS application server or the Web service container, or even directly upon the Java virtual machine. The running model of WS-CDL+ is shown in Figure 1 below.

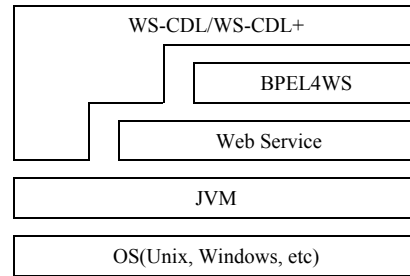


Figure 1. Running model of the WS-CDL+ execution engine

We now classify WS-CDL/WS-CDL+ as a kind of collaboration technology, the difference between the composition and collaboration technology are mainly originated from their applying situations. Collaborations work in the decentralized and federated environment, which is usually the case in B2B applications, while compositions work in the controller-centered mode, and are usually interested in one single participant, dealing mainly with the business processes of that participant.

Starting from this point, it is relatively easy to find the differences between these two technologies. However, one aspect that is still worth noting, is that collaborations work in the information driven mode, rather than explicitly invoking a certain Web service like compositions do. For Example, in a ticket-ordering scenario, customers may have the salesmen receive the money, query the ticket, print the ticket, then give the ticket, or just pay for the ticket and receive it. Actually, partner participants are usually unwilling to expose their business processes. They merely specify what to receive and what to send. What to do after reception is their decision.

Taking WS-CDL and BPEL4WS as examples, Table 1 presents the main differences between compositions and collaborations.

Table 1. Difference between WS-CDL and BPEL4WS

WS-CDL	BPEL4WS
The Collaboration layer specification	The Composition layer specification
Information driven	Explicitly invoking
Among participants	Within one single participant
Distributed controlling	Centralized controlling
Description language	Process execution language
Peer-to-peer	Centralized executor
Dynamic topology support	-
Candidate recommendation	Officially published
No commercial implementation	Websphere, ActiveBPEL, etc

3. WS-CDL+: extensions built into the execution engine

As stated previously, WS-CDL+ is based upon the WS-CDL specification by W3C. These extensions include variable initialization, user-defined functions, etc. One common criterion when applying these extensions is to keep compatible with WS-CDL.

1) Variable initialization. In WS-CDL, no specific mechanism is provided for variable initializations, leaving this to the *<assign>* action. Not only is this approach inconvenient and redundant to code, but also mixes the user data and the business logic altogether, making the code hard to maintain. WS-CDL+ solves this problem by adding the *initValue* attribute in *<variable>*, which goes like the following. When *initValue* is not specified, the variable is automatically initialized to NULL.

```
<variable name="NCName" .....
  initValue="WS-CDL+-Expression"? ...../>
```

2) User-defined functions. The user-defined functions supported in WS-CDL+ provide much more flexibility to its users. As a description language, computation power is no longer an important element, but is still essential. WS-CDL+ extends a new *<function>* tag to support the user-defined function mechanism, which is defined as follows. Definition I enables users to define a function using a WS-CDL+ expression, while Definition II converts a method in a certain Java class into a user-defined function.

```
Definition I:
<function name="NCName" returnType="QName"
  expression="WS-CDL+-Expression">
  <parameter name="NCName" type="QName" />*
</function>
Definition II:
<function name="NCName" returnType="QName"
  className="package.class"
  methodName="method">
  <parameter name="NCName"? type="QName" />*
</function>
```

3) Extended interaction method. WS-CDL+ extends WS-CDL's interaction method by introducing a new interaction model, namely NOTIFICATION, and two sub-tags, *<inputMessage>* and *<outputMessage>*, in the *<interaction>* tag. Conceptually, NOTIFICATION differs from both ONE-WAY and REQUEST-RESPONSE in that a) messages are sent from toRoleType to fromRoleType, b) the operation attribute in the *<interaction>* tag is no longer a must, and c) if the operation attribute is set, the Web service specified by it is invoked prior to the message exchange.

The introduction of the other two sub-tags is intended to give more flexibility to its users. By using these two tags, the WS-CDL+ participant is allowed to manually specify the message used to invoke the corresponding Web service, or the returning value sent back to its requester. Moreover, WS-CDL+ also specifies two implicit variables in each roleType instance, *cdl:wsIn* and *cdl:wsOut*, mainly to be used in the *<inputMessage>* and *<outputMessage>* tags. Actually, when NOTIFICATION is used with the operation attribute specified, it is essential to specify the *<inputMessage>* tag, or there is

no way for the invoked Web service to get its input parameters.

The definition of the extended *<interaction>* tag goes like the following:

```
<interaction .....>
  <participate .../>
  <exchange ...>
    <send .../>
    <receive .../>
  </exchange>*
  <timeout .../>
  <inputMessage>
    <part value="WS-CDL+-Expression" />+
  </inputMessage>?
  <outputMessage>
    <part value="WS-CDL+-Expression" />+
  </outputMessage>?
  <record ...>
    <source .../>
    <target .../>
  </record>*
</interaction>
```

4) Timer. Timer is one of the most important components in the modern programming languages. Most of the popular software developing technologies, such as J2EE and BPEL4WS, are all facilitated with this functionality. However, the WS-CDL specification by W3C does not give support to the timer mechanism, and it is a real pity! WS-CDL+ provides the built-in timer support by extending the *runIf* and *runWhile* attributes in *<choreography>*, as shown in the following. The difference of these two attributes goes like that of the *if* and *while* statements in C++/Java.

```
<choreography name="NCName"
  runIf="yyyy-mm-dd D hh:mm"|
  runWhile="yyyy-mm-dd D hh:mm"?
  ..... >
</choreography>
```

5) Debugging and exception. The debugging and exception handling ability is weak in the WS-CDL specification. WS-CDL+ extends these by adding three corresponding actions. The *<output>* action is used to print a string, a variable or a WS-CDL+ expression to the console, the *<assess>* action is used to evaluate a variable or an expression, and the *<raiseException>* action is used to cause a specified exception when the specified condition is satisfied. These actions are defined like as the following:

```
<output text="string"?|
  variable="QName"?|
  expression="WS-CDL+-Expression"?
  roleTypes="list of QName"? />
Definition I:
<assess variable="QName" value="any"?
  roleTypes="list of QName"? />
Definition II:
<assess expression="WS-CDL+-Expression"
  roleTypes="list of QName"? />
<raiseException name="QName"
  when="WS-CDL+-Expression"?
  roleTypes="list of QName"? />
```

6) Implicit finalization mechanism. In WS-CDL, a choreography must be explicitly finalized using

`<finalize>` against a defined `<finalizerBlock>` tag. However, we can require the execution engine to implicitly finalize a choreography instance according to a specified condition in WS-CDL+ by extending the `finalizeMechanism` attribute in `<choreography>` and the `when` attribute in `<finalizerBlock>`. When the implicit or mixed finalization mechanism is specified, the execution engine will automatically choose a `<finalizerBlock>`, whose condition (specified by the `when` attribute) is satisfied when the choreography is entering the successful state. This mechanism makes the finalization easier and more flexible, which goes like the following.

```

<choreography .....
  finalizeMechanism="implicate|explicate|mix"?
  ..... >
  .....
  <finalizerBlock name="NCName"
    when="xsd:boolean WS-CDL+-Expression"?>
    Activity-Notation
  </finalizerBlock>
</choreography>

```

4. Building the WS-CDL+ Execution Engine

WS-CDL+ execution engine is a WS-CDL+ runtime environment based upon Java and XML. Like many of the BPEL4WS execution engines, its target is to be implemented as a plug-in of such application servers as JBoss, Geronimo, or Tomcat.

At runtime, the execution engine will first load and parse the corresponding wsdl and cdl files, according to the instructions of the service deployment description file. Having parsed these files, the execution engine has created a set of correlated WS-CDL+ entities, such as the roleType entities and the choreography entities. The execution engine is then ready to accept incoming requests and create new choreography instances at that time. Once an instance is started, the engine processes in a strict peer-to-peer and message-driven mode according to the choreography description file. If the instance is completed successfully, the engine finalizes it using the implicit or explicit mechanism. However, if a fault occurs during execution, it will be propagated to all of the participants by the coordination mechanism, thus making them enter the fault handling process in synchronization.

Briefly speaking, the WS-CDL+ execution engine is mainly going to provide the following functionalities: a) parsing the WS-CDL+ description document into a set of correlated WS-CDL+ entities, b) supporting the system environment variables, c) providing both the built-in functions and the user-defined functions, d) creating and initializing a choreography instance by the user request, the `<perform>` activity, the incoming message and the timer, e) supporting a variety of underlying communication protocols and three kinds of communication styles, f) giving support to the dynamic service by passing channels, g) implementing the implicit

and explicit finalization mechanism, h) fault handling, i) debugging, and j) coordination.

4.1. Initiation and lifecycle

A WS-CDL+ choreography instance can be created by four approaches: 1) the user request, 2) the `<perform>` activity, 3) the incoming message, and 4) the timer. They act like the following:

1) By the user request. The execution engine will create a choreography instance according to the instructions from the user program or the WS-CDL+ deployment description file. In this approach, the root attribute of the corresponding choreography must be true.

2) By the `<perform>` activity. During the execution of a choreography instance, the engine will be creating a new choreography instance when executing a `<perform>` activity. At this time, if the block attribute in `<perform>` is true, the newly created instance will be executing in the same thread as the parent instance; if false, the engine will create a separated thread at the same time, and make it execute in that thread.

3) By the incoming message. When the execution engine receives a message through a channel, its identities will be first extracted and analyzed according its corresponding channelType definition. If it is found to be sent by the initiator of a newly created instance, the engine will automatically create a new instance and transmit the received message into that instance.

4) By the timer. If the extended attribute, `runIf` or `runWhile`, is specified in `<choreography>`, the engine will create and initialize a new instance when either attribute is matched.

Once created, the instance will automatically be in the NEW status, and begin to execute the initialization process. During this period, the execution engine will create roles and variables according to the definition of the choreography, and correlate the variables to the corresponding roles. The instance then enters the ENABLED state when the initialization process is done, in which state it is capable of sending/receiving messages to/from its related participants and proceedings, according to the choreography definition.

The instance in the ENABLED state must be unsuccessfully completed when an exception is caused during the execution, and the `exceptionBlock`, if present, will be enabled, thus causing the instance to enter the UNSUCCESSFUL state. This instance will then enter the CLOSED state when the `exceptionBlock` is completed. However, if there is no `exceptionBlock` existing, the instance will implicitly enter the CLOSED state and the exception occurred will be propagated to its parent instance, if present.

Alternatively, the instance must be completed successfully if its complete condition is matched. The

execution engine calculates the value of the complete attribute after completion of each activity. When this happens, the instance enters the SUCCESSFUL state, and the rest of the activities are disabled, except for any *finalizerBlocks*. The instance in the SUCCESSFUL state will enter the CLOSED state once one of its installed *finalizerBlocks* is enabled and completed, either when implicated or explicated applied, or it has no *finalizerBlocks* defined.

4.2. Communication model

Unlike that of WSFL or BPEL4WS, the WS-CDL/WS-CDL+ participants communicate in a peer-to-peer way. It is a completely new model, with the following characteristic: a) peer-to-peer communication between participants, b) message-driven collaboration by conveying variable values, c) various underlying communication protocols such as HTTP, FTP and SMTP, d) automatic message dispatch using the identity definition in channelType, e) dynamic service support by passing channels, and f) exception propagation mechanism with the built-in coordination facility.

In this model, messages are divided into three types: value messages, exception messages, and coordination messages. A value message is actually a serialized variable value. It is the most frequently used message type, used to drive the progress of choreography. Messages such as ticket order request and payment confirmation all belong to this type. An exception message is used to convey the exception information between coordinators. It is crucial for the coordination facility among the WS-CDL+ participants, and its behaviors are all controlled by the execution engine within the coordination channels. It is transparent to the users. The coordination message is used to convey the coordinator addresses among participants, and is also transparent to the users. We will further discuss the use of the exception and coordination message type in Section 4.3.

Table 2. Communication style simulations by HTTP and SMTP

	HTTP	SMTP
One-way	A HTTP Post action from sender to receiver that returns an empty payload	Request by sending a SMTP packet
Request-response	A HTTP Post action from sender to receiver that returns another packet back to sender	Request by sending a SMTP packet, and response by sending another SMTP packet back to the sending endpoint
Notification	A HTTP Post action from receiver to sender that returns an empty payload	A SMTP packet from receiver to sender

There are three basic communication styles in WS-CDL+: one-way, request-response, and notification. However, the underlying communication protocols such as HTTP, FTP, and SMTP, upon which this communication model is built, cannot support all of the three styles. To solve this problem, we have to simulate these communication styles using the underlying protocols. Table 2 takes HTTP or SMTP as examples to demonstrate the simulations.

A message channel in WS-CDL+ is composed of a channel-way in the sender and an endpoint in the receiver, as shown in Figure 2. The channel-way in the sender is a variable and message passing path. As a variable, it stores the address of the endpoint in the receiver, while as a message passing path, it can be used to send or receive messages. The endpoint in the receiver is mainly composed of the receiving point and the message dispatcher, which is used to dispatch a received message to its corresponding role. It also provides a protocol-independent abstract transmission layer to the receiver.

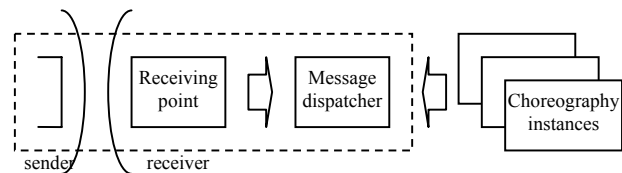


Figure 2. Structure of the message channel

Upon the reception of a message from the receiving point, the message dispatcher extracts the identities of the message using the identity extractor and the choreography instance index, which maps the message identities to the pointers to the message pipelines, sends the message to the pipeline, and returns. After that, it is up to the consummating choreography instance to fetch it from the pipeline and store into the appropriate variable. However, when the consummating choreography instance is fetching a message from the pipeline while there is no message in the pipeline, the instance will be waiting until successfully read or timed out.

4.3. Coordination

Coordination means that during the execution of choreography, if an exception occurs in any one of the participants, it must be propagated to all the participants, so that they can terminate it the same way. In the WS-CDL+ execution engine, the software entity providing the coordination facility is called the coordinator. Each of the participating instances has its own coordinator.

To send the coordination message to another participating instance, the sender must have the endpoint address of its counterpart. A coordinator stores the addresses of those whose instances have joined in the choreography, and two coordinators who have acquired

the addresses of their counterparts are considered to have created a connection between them.

The address table within the coordinator is created like the following. When a choreography instance is initialized, the execution engine will automatically create a corresponding coordinator for it, and add the coordinator address of its own into the address table, so that the address table of the coordinator stores its own address only. After that, when a message exchange happens between two roles, their coordinators also send their whole address tables to the counterparts. Then the instances will acquire the address tables and hand them over to their own coordinators after the message processing is done. By doing this, the coordinator can record all of those who have communicated with it, and when an exception occurs, the coordinator can use its address table to propagate the exception, thus fulfilling coordination mechanism.

Internally, the execution engine employs a special type of message, the coordination message, to transfer the address table, while the exception itself is propagated using another message type, the exception message.

5. Experimentation

In order to test the functionality and performance of the execution engine, we developed a ticket ordering use case, and used the prototype system. The PC used to run this use case is configured as follows: Pentium 4 2.66GHz HT with 512M RAM, Windows XP Professional, Java Standard Edition 5.0 and Eclipse 3.1.1.

5.1. Functional experiments

Functional tests focused on the function aspect of the execution engine. It is used to ensure that the WS-CDL+ execution engine works properly according to the WS-CDL+ document. The use case involves three participants, TravelService, AirTicketService, and BankService. At the beginning of the choreography, TravelService first sends a ticket subscribing message to AirTicketService. AirTicketService receives the message, checks whether the designated ticket is available, and returns the checking result back to TravelService. If check passed, AirTicketService notifies TravelService to make payment at BankService. When payment is made, BankService sends a confirmation message to AirTicketService, and AirTicketService then sends the electronic ticket to the TravelService, which is represented as the ticket information in this use case. Figure 3 and Figure 4 print some of the critical variable values when TravelService receives payment notice from AirTicketService, and the electronic ticket, represented as the ticket information, after the order is made.

```

<corunkit name="w1" guard="cdl:isVariableAvailable('spi:bankRole') or cdl:getVariable('spi:
<sequence>
  <interaction name="PaymentNoticeToTravel" operation="QueryTicketPrice" channelVariable=
  <participate relationshipType="spi:Travel-Ticket" fromRoleTypeRef="spi:TicketRole" to
  <exchange name="request" informationType="spi:PaymentNoticeType" action="notify">
    <send variable="cdl:getVariable('spi:paymentNotice','')"/>
    <receive variable="cdl:getVariable('spi:paymentNotice','')"/> recordReference="rec
  </exchange>
  <inputMessage>
    <part name="ticketID" value="cdl:getVariable('spi:ticketSubscribeRequest','id','')
  </inputMessage>
  <outputMessage>
    <part name="bankURI" value="cdl:getVariable('spi:bankChannel','')"/>
    <part name="ticketID" value="cdl:getVariable('spi:ticketSubscribeRequest','id','')
    <part name="receiverAccount" value="123-456-789-x"/>
    <part name="amount" value="cdl:getVariable('cdl:waOut','price','')"/>
  </outputMessage>
  <record name="recordBankChannel" when="after">
    <source variable="cdl:getVariable('spi:paymentNotice','bankURI','')"/>
    <target variable="cdl:getVariable('spi:bankChannel','')"/>
  </record>
</interaction>

```

Figure 3. Some variable values when payment notice received from AirTicketService

```

<corunkit name="w2" guard="cdl:isVariableAvailable('spi:travelRole') or cdl:getVariab
<sequence>
  <interaction name="TicketOrdered" operation="GetTicketInformation" channelVariabl
  <participate relationshipType="spi:Travel-Ticket" fromRoleTypeRef="spi:TicketRo
  <exchange name="transferTicketInformation" informationType="spi:TicketInformati
    <send variable="cdl:getVariable('spi:airTicket','')"/>
    <receive variable="cdl:getVariable('spi:airTicket','')"/>
  </exchange>
  <inputMessage>
    <part name="id" value="cdl:getVariable('spi:ticketSubscribeRequest','id','')
  </inputMessage>
  </interaction>
  <output variable="cdl:getVariable('spi:airTicket','')"/> roleTypes="spi:TravelRo

```

Figure 4. Electronic ticket received

5.2. Performance experiments

Next, we will approximately test the running performance between Java and WS-CDL+ by measuring the time they spend repeatedly executing a certain interaction task. In Java, we use two thread to simulate the participants of WS-CDL+, and use Java code to represent the interaction process of these two participants. The experimental results are shown in Table 3 (in millisecond) and Figure 5:

Table 3. Experimental results on running performance between Java and WS-CDL+

Loop Count	100	200	300	400	500
WS-CDL+	1304	1938	2514	3075	3554
Java	569	1030	1363	1676	1970
Loop Count	600	700	800	900	1000
WS-CDL+	4073	4577	5276	5669	6058
Java	2296	2605	2871	3223	3524

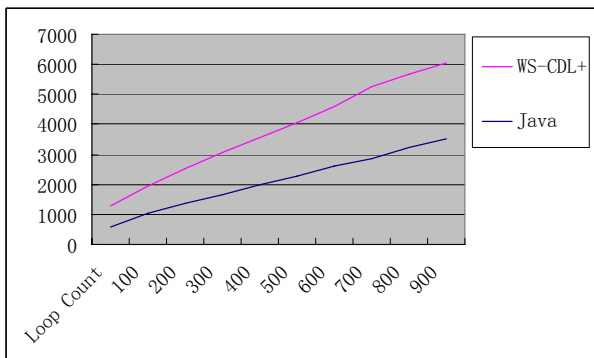


Figure 5. Experimental results of running performance

5.3. Discussion

From the experiments above, we can see that the execution engine we implemented is capable of completing the functionalities of WS-CDL+, fulfilling collaboration among participants, and supporting dynamic service. As for its running performance, it is able to complete an interaction in approximately 6 to 10 milliseconds in our experimental environment, which is close to packet transmission speed over Internet, and therefore in accordance with its application requirements. As a description language, it depends on Java as its underlying interpreting language, so its performance is surely lower than Java. However, from the second experiment in Section 5.2, we can see that the time consumed for WS-CDL+ to run a specific interaction is not more than 2 times that of Java. Therefore, we consider this amount of performance loss is worthwhile and acceptable, since WS-CDL+ greatly improves the quality and maintainability of the code.

6. Related work

The research of Web service composition/collaboration can be classified into two catalogues: methods based on Artificial Intelligence and methods based on flows. As for WS-CDL, the research includes its relationship with BPEL4WS, formal semantics and verification, and so on.

In one hand, the methods based on Artificial Intelligence include semantic Web and AI planning, which feature automatic or semi-automatic composition. OWL-S [13] and WSMO [14] are two semantic Web methods, while the Hierarchical Task Network (HTN) planning [15] is one of the main AI planning technologies.

In the other, the methods based on flows mainly consist of practical technologies and theoretical models. The practical technologies include WSFL [16], WSCI [18], XLANG [17], BPEL4WS [19], WS-CDL [1][2][3][4]. BPEL4WS is good in supporting service

composition within a certain participant; however, it is weak in supporting dynamic service, service collaboration between peers, and verification of service composition, all of which make it not a best fit for service collaboration among peer participants. WS-CDL is based on Pi-calculus, whose idea is derived from WSCI, and has become a W3C candidate recommendation since November, 2005. WS-CDL works at the collaboration layer of the emerging Web service stack, and is targeted at describing the collaboration relationship between peers. The theoretical models include Pi-calculus [12], Petri net [7], etc. They are based upon the mathematical theory, and are more concise in expression. Their theoretical nature makes them good toolkits for composition analysis and verification; however, it also prevents them from being directly used in the application fields, thus hard to be acceptable by common developers.

The global calculus and the end-point calculus [9], proposed by the W3C Web Service Choreography Working Group, presents a theoretical basis of communication-centered, concurrent programming. The global calculus originates from WS-CDL, while the end-point calculus is a typed Pi-calculus. CDL [10] is a formal model of the simplified WS-CDL, which includes important concepts related to participant roles and collaborations among them in a choreography [10], and models choreography with a set of participant roles and the collaboration among them [10]. After translating WS-CDL to the input language of the model-checker SPIN [21], we can automatically verify the correctness of a given choreography [10]. The relationship between WS-CDL and BPEL is also a research concern of WS-CDL. Mendling and Hafner [20] propose a translation approach and implement a transformation program, *wscdl2bpel.xslt*, capable of generating BPEL stubs from WS-CDL.

7. Conclusion

The prototype system of the WS-CDL+ execution engine we discussed is the first attempt in bringing the WS-CDL documents into execution. It enables the WS-CDL researchers to test and evaluate their jobs, and provides an execution environment for the developers to run their WS-CDL/WS-CDL+ documents in a simulative environment. Moreover, we can also turn it into a plug-ins for J2EE application servers such as JBoss or Tomcat with only minimal modifications.

The extended features of WS-CDL+ that are built into the execution engine include the user-defined functions, the timers, the extended interaction model and the implicit finalization mechanism. These extended functionalities enable developers to write more concise, readable, and maintainable WS-CDL+ documents, and easily realizing the functionalities that are hard or even impossible to implement with WS-CDL.

In all, with the development of SOC and Web service, deploying a massive Web service collaboration system among participants will surely be the next hotspot of SOC, and WS-CDL/WS-CDL+ is providing a solid basis for this development. It can be foreseen that as its execution engine goes more capable, WS-CDL will surely become an ideal solution to implement business collaborations between peers.

Acknowledgements

This work is supported in part by NSFC with grant no. 60473091 and 60673175.

References

- [1] D. Austin, A. Barbir, E. Peters, and S. Ross-Talbot, *Web Services Choreography Requirements*, <http://www.w3.org/>, 2004.
- [2] D. Burdett, and N. Kavantzias, *WS Choreography Model Overview*, <http://www.w3.org/>, 2004.
- [3] N. Kavantzias, D. Burdett, G. Ritzinger, T. Fletcher, Y. Lafon, and C. Barreto, *Web Services Choreography Description Language Version 1.0*, <http://www.w3.org/>, 2005.
- [4] S. Ross-Talbot, and T. Fletcher, *Web Services Choreography Description Language: Primer*, <http://www.w3.org/>, 2006.
- [5] C. Yushi, L.E. Wah, and D.K. Limbu, "Web Services Composition - An Overview of Standards", *Synthesis Journal*, iTSC, Singapore, 2005, pp. 137-150.
- [6] N. Milanovic and M. Malek, "Current Solutions for Web Service Composition", *IEEE Internet Computing*, USA, November & December 2004, pp. 51-59.
- [7] R. Milner, J. Parrow, and D. Walker, "A Calculus of Mobile Processes, Part I and Part II", *Information and Computation*, 1992, Volume. 100, Issue. 1, pp. 1-77.
- [8] J. Liao, H. Tan and J.D. Liu, "Describing and Verifying Web Service Using Pi-Calculus", *Chinese Journal of Computers*, China, April 2005, pp. 635-643.
- [9] M. Carbone, K. Honda, N. Yoshida, R. Milner, G. Brown, and S. Ross-Talbot, *A Theoretical Basis of Communication-Centred Concurrent Programming*, <http://www.w3.org/>, 2006.
- [10] H.L. Yang, X.P. Zhao, and Z.Y. Qiu. "A Formal Model for Web Service Choreography Description Language (WS-CDL)", *International Conference on Web Services 2006 (ICWS 2006)*, Chicago, USA, Sep. 2006.
- [11] M.P. Papazoglou, and D. Georgakopoulos, "Service-Oriented Computing", *Communications of the ACM*, October 2003, pp. 25-65.
- [12] C.A. Petri, *Kommunikation mit Automaten*, PhD thesis, Institut fuer Instrumentelle Mathematik, Bonn, 1962.
- [13] D. Martin, M. Burstein, J. Hobbs, O. Lassila, D. McDermott, S. McIlraith, S. Narayanan, M. Paolucci, B. Parsia, T. Payne, E. Sirin, N. Srinivasan, and K. Sycara, *OWL-S: Semantic Markup for Web Services*, <http://www.daml.org/>, 2004.
- [14] The ESSI WSMO working group, *Web Service Modeling Ontology*, <http://www.wsmo.org/>.
- [15] D. Nau, T.C. Au, O. Ilghami, U. Kuter, J. William, D. Wu, and F. Yaman, "SHOP2: An HTN Planning System", *Journal of Artificial Intelligence Research*, December 2003, pp. 379-404.
- [16] F. Leymann, *Web Services Flow Language (WSFL) Version 1.0*, <http://www-3.ibm.com/>, 2001.
- [17] S. Thatte, *XLANG*, <http://www.gotdotnet.com/>, 2001.
- [18] A. Arkin, S. Askary, S. Fordin, W. Jekeli, K. Kawaguchi, D. Orchard, S.o Pogliani, K. Riemer, S. Struble, P. Takacsi-Nagy, I. Trickovic, and S. Zimek, *Web Service Choreography Interface (WSCI) 1.0*, <http://www.w3.org/>, 2002.
- [19] A. Arkin, S. Askary, B. Bloch, Y. Golland, N. Kartha, C.K. Liu, S. Thatte, P. Yendluri, and A. Yiu, *Web Services Business Process Execution Language Version 2.0*, <http://www.oasis-open.org/>, 2004.
- [20] J. Mendling and M. Hafner. "From Inter-Organizational Workflows to Process Execution: Generating BPEL from WS-CDL", *OTM 2005 Workshops*, Lecture Notes in Computer Science 3762, Springer Verlag, October 2005, pp. 506-515.
- [21] G.J. Holzmann, *The SPIN Model Checker: Primer and Reference Manual*, Addison-Wesley, 2003.