

WS-CDL+ for web service collaboration

Zuling Kang, Hongbing Wang

School of Computer Science and Engineering

Southeast University

P.R.China

Patrick C. K. Hung

Faculty of Business and Information Technology

University of Ontario Institute of Technology

Canada

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 also a crucial component to put WS-CDL into the application field. This paper implements an execution engine for WS-CDL, which has never been built before, and enhance its functionalities and usability by introducing six extended features into the execution engine, namely WS-CDL+. Several experiments are then done to test the functionalities and performance of the execution engine, making sure that it works properly according to the WS-CDL+ specification and its performance is acceptable. Finally, the whole paper is concluded, as well as the discussion of the application perspectives of WS-CDL/WS-CDL+.

Keywords: WS-CDL+, Web service, Execution engine, Collaboration

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. Web services are autonomous software systems identified by URIs, which can be advertised, located, and accessed through messages encoded according to XML-based standards, and transmitted using Internet protocols (Austin, Barbir, Ferris & Garg, 2004). They are designed to support interoperable machine-to-machine interaction over a network, and are now becoming the main building blocks of SOC (Brown & Haas, 2004). SOC is the computing paradigm that utilizes services as fundamental elements for developing applications/solutions (Ge3orgakopoulos & Papazoglou, 2003). It stands on a higher level, and creates software systems by composing the elementary services, mainly in form of Web services, with service composition technology such as WSFL, XLANG, or BPEL4WS (Leymann, 2001; Thatte, 2001; Arkin, et al., 2004). Because Web services are based on these open and widely-accepted protocols, such as HTTP and SOAP, and provide a uniform and ubiquitous information distributor for a wide range of computing devices and software platforms, they, as

well as SOC, constitute the next major step in distributed computing (Lafon & Mitra, 2007).

The composition technologies, which define an executable process to be enacted by a central orchestration engine, and are a single participant's view of the world, such as WSFL or BPEL4WS. They address one aspect of the SOC, the collaboration technologies, which describe an interaction protocol to be enacted by the peers without any central entity governing the collaboration, and providing a global view of the peers, which address the other aspects of the SOC. WS-CDL is one of the most promising collaboration technologies in the service computing area.

One of the biggest problems going with WS-CDL is that it lacks a real or even prototype execution engine for it to run. Nevertheless, further researches of WS-CDL and its applications do need such an execution engine to experiment on their research conclusion, or test their new properties, or create a B2B message collaboration system. Motivates are then initiated to implement such an engine. However, when building and testing the execution engine, we find it necessary to add extended features to the current WS-CDL specification, so as to enhance the functionalities of WS-CDL itself, and also make it easy and concise for developers to express their collaboration processes. These extensions finally leads to the formation of WS-CDL+. Based upon WS-CDL, WS-CDL+ mainly provides 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 from different WS-CDL/WS-CDL+ implementers as well.

Our work aims to advance the current state of the Web service collaboration technology by addressing the following two issues: **a) A 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 on, 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.

There are mainly two issues for consideration during the design of the execution engine, one of which is how to parse a cdl file, while the other is how to design the communication model between WS-CDL+ participants.

Building a parser that creates a set of correlated CDL entities out of the cdl file is the first step to a running execution engine. Since a cdl file an XML document, we consider implementing a cdl parser based upon an XML parser, such as JDOM or DOM4J. After paring a cdl file into a set of XML entities, we set out to turn these XML entities into CDL entities. During that courses, operator precedence parsing method is chosen to handle WS-CDL+ expressions, while LL(1)

parsing method is used to deal with WS-CDL+ activities.

When it comes to the design of the communication model, there are still many problems to solve, including how to define the format of messages to exchange, how to simulate three basic communication styles of WS-CDL+, namely one-way, request-response and notification, using the underlying communicating protocols, how to design the message channel and how to deal with coordination problems. We will further address these problems in Section 4. Here, we just illuminate that the communication model of WS-CDL+ can be applied to other distributed computing systems, not only 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 six aspects of extension mechanisms that are built into our execution engine. 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 whole text.

2 Preliminaries

WS-CDL 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 (Austin, Barbir, Peters, & Ross-Talbot, 2004; Burdett & Kavantzias, 2004; Barreto, Burdett, Fletcher, Kavantzias, Lafon & Ritzinger, 2005; Fletcher & Ross-Talbot, 2006). It describes this behavior from a global or neutral perspective rather than from the perspective of any one party. It is worth pointing out that the Web Services Choreography Working Group of W3C is still working on this specification and it has not finally come into being; however, it has been a W3C candidate recommendation since Nov. 9, 2005.

Since WS-CDL is not officially published, it is worthwhile to discuss the improvements and extensions of it, so as to make it a more suitable fit in applications. This is the initial motivator of WS-CDL+ that we will discuss. WS-CDL+ extends WS-CDL by adding features such as user-defined functions, and timers and console outputs, while imposing restrictions to what the WS-CDL specification has not explicitly stated. This increases the co-operability as well as reduces the arbitrariness of different WS-CDL/WS-CDL+ implementations.

2.1 Emerging Web service protocol stack and WS-CDL/WS-CDL+

The classical Web service protocol stack focuses on a single Web service, and defines four layers, which, from bottom to top are, the Transport layer, the Message layer, the Description layer, and the Discovery layer. By taking advantage of these four layers and the protocols that are defined in each layer, developers can easily find and invoke their expected Web services.

Since the classical Web service protocol stack appears in the early stage of the SOC era, it merely concerns the basic functionalities of a single Web service. Many advanced properties, especially transactions, compositions, and collaborations, are not addressed. So, as the development of the SOC technology continues, it can never meet the application requirements.

Thus by extending this classical Web service protocol stack, we get the emerging Web service protocol stack, as shown in Figure 1:

Collaboration	WS-CDL		
Composition	WSFL, BEPL4WS		
Quality of Service	Reliable	Security	Transaction
	Messaging		Coordination

Discovery	UDDI		
Description	WSDL		
Messaging	SOAP		
Transport	HTTP, SMTP, JMS		

Fig. 1. Emerging Web service protocol stack

From the above figure, we can see that the emerging Web service protocol stack adds three layers upon the classical Web service protocol stack, which are the QoS layer, the Composition layer, and the Collaboration layer. The QoS layer is mainly used to ensure that the provider and consumer of a Web service will communicate correctly and provide the transaction support for the upper two layers. The Composition layer provides the orchestration mechanism, enabling a single participant to recursively define a new service by composing existing services, and presenting the result as a service. One of the most famous workflow technologies, BPEL4WS, is defined within this layer.

The topmost layer of the emerging Web service protocol stack is the Collaboration layer. It is built upon the Composition layer and is used to process the choreography of Web services. Choreography differs from orchestration in that choreography is concerned with global, multi-participants, and peer-to-peer collaboration, without any need for centralized controllers, while orchestration focuses on the behavior of a single participant, and depends on a centralized controller. WS-CDL, as well as WS-CDL+ in this paper, consists of only the technology in the Collaboration layer. As a kind of technology in this layer, it describes the behavior of multi-participants in a global view, operates in a peer-to-peer mode, and makes use of services in the Composition layer, such as invoking a composite Web service to fulfill its assignments.

2.2 Purpose and goals

The WS-CDL specification is aimed at being able to precisely describe collaborations between any type of participant regardless of the supporting platform or programming model used by the implementation of the hosting environment (Barreto, et al., 2005). More specifically, its primary goal is to specify a declarative, XML-based language that defines, from a global viewpoint, the common and complementary observable behavior; specifically, the information exchanges that occur and the jointly agreed ordering rules that need to be satisfied (Barreto, et al., 2005).

When discussing the necessity of WS-CDL, one commonly asked question is whether WS-CDL can be replaced by the Composition layer technologies, such as BPEL4WS. Of course, we have to admit that there does exist similarities in the functionalities between WS-CDL and the Composition layer technologies. However, it is not wise to explicitly invoke the Web services of the partner participants. It is possible for your partners to change the locations or the service names, and this will lead you to change your process definitions if you're using BPEL4WS.

2.3 The WS-CDL+ model

The program model of WS-CDL+ is built upon that of WS-CDL (Barreto, et al., 2005). 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 WS-CDL.
- workunit. The same as that WS-CDL.
- activities and ordering structures. Adding three new basic activities, *<output>*, *<assess>* and *<raiseException>*, to support the debug and exception functionality.
- interaction activity. The same as that WS-CDL.
- semantics. The same as that WS-CDL.

The WS-CDL+ execution engine is built using the Java programming language, and executing on top of the BPEL4WS application server, or directly upon the Java virtual machine, the Web service container, which is shown in Figure 2.

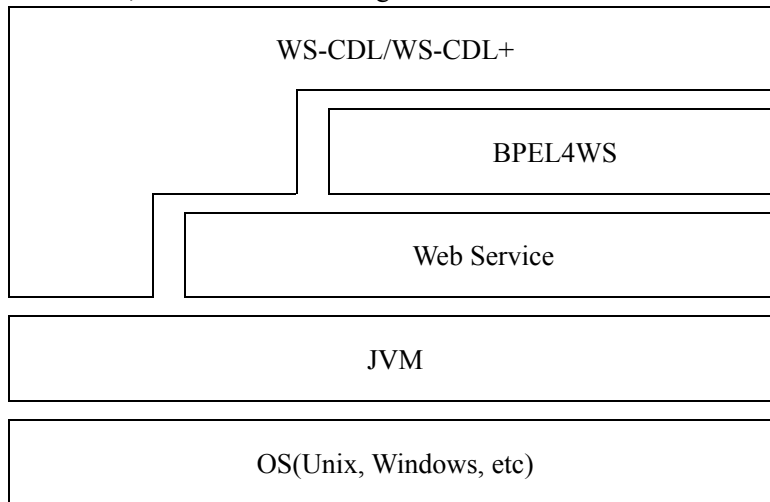


Fig. 2. The running model of the WS-CDL+ execution engine

2.4 Composition VS. Collaboration

When discussing the difference between the composition and the collaboration, it is worthwhile to note that most of these differences are originated from their applying situations. Collaborations are used in situations where there are many participants, and the information exchanges among these participants are the areas of concern. Compositions, on the other hand, are usually interested in one single participant, and mainly deal with the business processes of that participant. However, the fact that compositions focus on a single participant does not mean that the Composition layer technologies cannot exchange information with other participants. But the amount of information exchanged is low in composition, and only serves the execution of a business process.

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

Despite these differences, in general, collaborations can be regarded as a special kind of composition. The purpose of compositions is to complete a complex task using some simple-function services; although this goal can also be realized by collaborations.

3 WS-CDL+: Improvements and Extensions to WS-CDL

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 maintain compatibility with WS-CDL, viz. For Example, a legal WS-CDL document must also be a legal WS-CDL+ document. In this section, we probe why and how WS-CDL+ extends WS-CDL, and provides the definitions of these extensions.

3.1 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 it also mixes the user data and the business logic together, making the code hard to maintain. WS-CDL+ solves this problem by introducing a specific variable initialization mechanism, adding the `initValue` attribute in the `<variable>` tag, like the following:

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

However, to keep compatible with WS-CDL, WS-CDL+ will automatically initialize a NULL value to the variable when the `initValue` attribute is not specified.

3.2 User-defined functions

The user-defined functions supported in WS-CDL+ provide much more flexibility to its users. WS-CDL defines a set of built-in functions and allows its execution implement to extend the function set. However, it does not present a mechanism for the user to add functions on demand. As a description language for service collaboration, 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:

```
<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>
```

There are two approaches to declare user-defined functions in WS-CDL+. Definition I is used to define a function using a WS-CDL+ expression, while Definition II is used to convert a method in a certain Java class into a user-defined function.

3.3 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.

Extended interaction model introduced, the definition of the *<interaction>* tag goes like the following:

```

<interaction name="NCName" channelVariable="QName"
  operation="NCName"?
  align="true|false"? initiate="true|false"?>
<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>

```

3.4 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>*, defined as the following:

```

<choreography name="NCName "
  runIf="yyyy-mm-dd D hh:mm"|runWhile="yyyy-mm-dd D hh:mm"?
  complete="xsd:boolean WS-CDL+-Expression"?
  isolation="true|false"?
  root="true|false"?
  coordination="true|false"? >
  .....
</choreography>

```

3.5 Debugging

The debugging mechanism is absent in the WS-CDL specification. WS-CDL+ extends this facility by adding two corresponding actions, *<output>* and *<assess>*. The action *<output>* is used to print a string, a variable value, or a WS-CDL+ expression value to the console, through which developers can monitor the variable values or others as they need. The syntax of *<output>* goes like the following:

```

<output text="string"?|
  variable="QName"?|
  expression="WS-CDL+-Expression"?
  roleTypes="list of QName"? />

```

Another extended action, *<assess>*, is used to evaluate a variable or a WS-CDL+ expression. The execution engine will automatically throw an exception named *cdl:assessFailure* when the evaluated variable is null. Its value is not equal to the expected value, or the evaluated WS-CDL+ expression is false. The syntax of it is defined like the following:

```

Definition I:
<assess variable="QName" value="any"?
      roleTypes="list of QName"? />

Definition II:
<assess expression="WS-CDL+-Expression"
      roleTypes="list of QName"? />

```

3.6 Exception

WS-CDL+ extends the exception mechanism of WS-CDL by introducing *<raiseException>*. This action is used to cause a specified exception when the specified condition is satisfied. However, if the *when* attribute is not set, this action will always cause the specified exception once it is executed. The *<raiseException>* action is defined as the following:

```

<raiseException name="QName"
      when="WS-CDL+-Expression"?
      roleTypes="list of QName"? />

```

3.7 Implicit finalization mechanism

In WS-CDL, a choreography must be explicitly finalized using *<finalize>* against a defined *<finalizerBlock>* tag, which is called explicit finalization mechanism in WS-CDL+. WS-CDL+ introduces another finalization mechanism, namely implicit finalization mechanism, which makes the finalization of a choreography easier and more flexible.

By having the implicit finalization mechanism works introduced, we can choose to apply the implicit, explicit, or mixed finalization mechanism by setting the extended *finalizeMechanism* attribute of *<choreography>*. 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. The following is the definition of *<choreography>* with WS-CDL+ finalization mechanism extended:

```

<choreography name="NCName"
  runAt="yyyy-mm-dd D hh:mm" | runWhen="yyyy-mm-dd D hh:mm"?
  complete="xsd:boolean WS-CDL+-Expression"?
  finalizeMechanism="implicate|explicate|mix"?
  isolation="true|false"?
  root="true|false"?
  coordination="true|false"? >
  .....
  <finalizerBlock name="NCName"
    when="xsd:boolean WS-CDL+-Expression"?>
    Activity-Notation
  </finalizerBlock>
</choreography>

```

However, to keep compatible to WS-CDL, the *finalizeMechanism* attribute defaults to

explicate when it is unspecified.

4 Building the WS-CDL+ Execution Engine

4.1 Overview of the 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.

This paper implements a prototype system of the WS-CDL+ execution engine and has implemented all of the functionalities of WS-CDL+. It can easily be turned into a plug-in for such application servers as JBoss, only by modifying it according the specific runtime environment.

Briefly speaking, the WS-CDL+ execution engine is mainly going to provide the following functionalities:

- ✧ Parsing the WS-CDL+ description document into a set of correlated WS-CDL+ entities
- ✧ Supporting the system environment variables
- ✧ Providing both the built-in functions and the user-defined functions
- ✧ Creating and initializing a choreography instance by the user request, the *<perform>* activity, the incoming message, and the timer
- ✧ Executing in a strict peer-to-peer mode
- ✧ Supporting a variety of underlying communication protocols and three kinds of communication styles
- ✧ Giving support to the dynamic service by passing channels
- ✧ Implementing the implicit and explicit finalization mechanism
- ✧ Fault handling
- ✧ Debugging
- ✧ Coordination

4.2 System architecture

The overall architecture of the WS-CDL+ execution engine is shown in Figure 3:

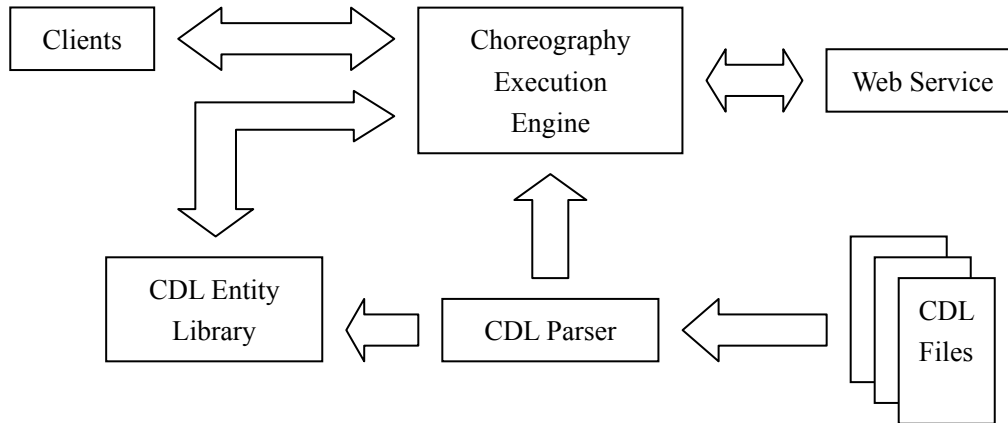


Fig. 3. Overall architecture of the WS-CDL+ execution engine

The main functionalities of the components are:

CDL parser: builds a correlated set of CDL entities using the WS-CDL+ description files, providing for use of the choreography execution engine.

CDL entity library: is used to store the correlated set of CDL entities such as roleTypes, channelTypes, choreographies, and activities. It is built during the initializing period, and used to provide choreography information during execution.

Choreography Execution Engine:

4.3 Initiation and lifecycle

As discussed in Section 4.1, 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, or it has no *finalizerBlocks* defined.

The lifecycle of a WS-CDL+ choreography instance is shown in Figure 4.

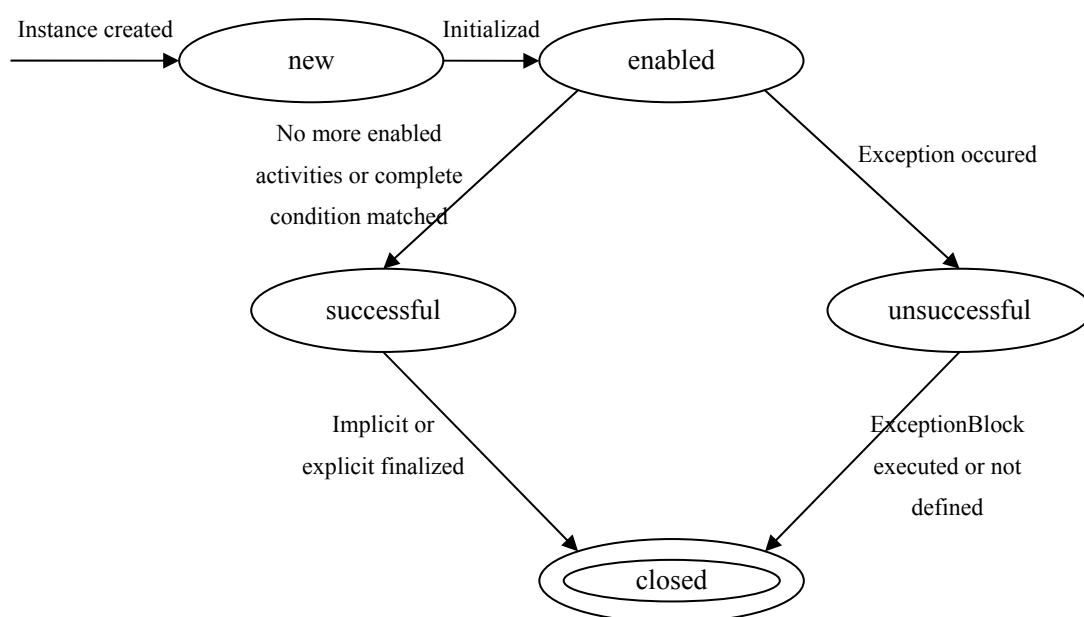


Fig. 4. The lifecycle of a WS-CDL+ instance

4.4 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:

- ✧ Peer-to-peer communication between participants
- ✧ Message-driven collaboration by conveying variable values
- ✧ Various underlying communication protocols such as HTTP, FTP, SMTP, etc
- ✧ Automatic message dispatch using the identity definition in channelType
- ✧ Dynamic service support by passing channels
- ✧ 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 sending and receiving are all controlled by the

execution engine within the coordination channels. It is transparent to the users. The third message type, the coordination message, is used to convey the coordinator addresses among participants, and is also transparent to the users, whose sending and receiving are controlled by the execution engine. We will further discuss the detail of the exception and coordination message type in Section 4.5.

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.

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

Another issue of the WS-CDL+ communication model is the implementation of the message channel. In the execution engine, a message channel is an instance of the channelType, composed with a channel-way in the sender and an endpoint in the receiver, as shown in Figure 5. 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. By using this channel-way mechanism, the execution engine provides a protocol-independent abstract transmission layer to the sender party. 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, or roleType instance. It also provides a protocol-independent abstract transmission layer to the receiver. When a data packet arrives, it first picks up the payload from its underlying protocol stack, deserializes it into a variable value, and hands it over to the message dispatcher for further processing.

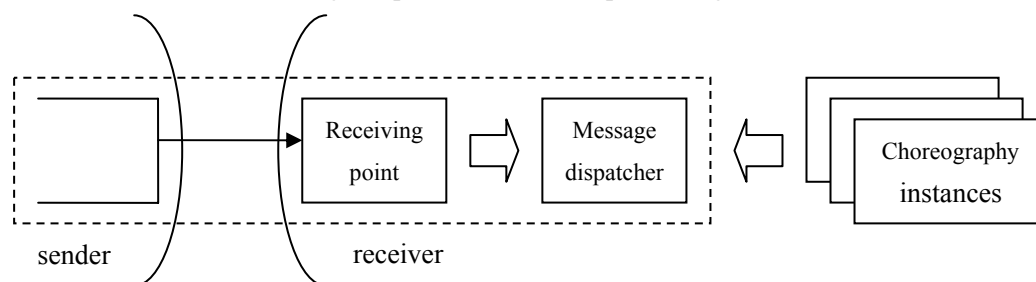


Fig. 5. The structure of the message channel

A message dispatcher is used to dispatch a received message to its corresponding role, based upon the identities in the message. Its structure is shown in Figure 6:

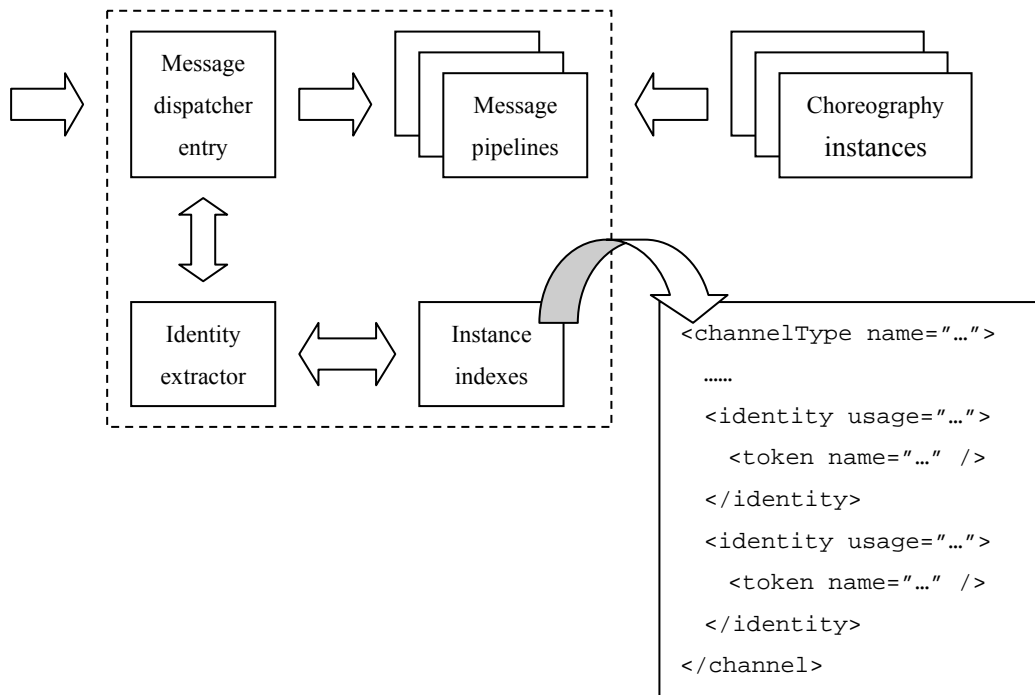


Fig. 6. The structure of the message dispatcher

Upon the reception of a message from the receiving point, the message dispatcher will first extract the identities of the message using the identity extractor and the choreography instance index, which maps the message identities of the pointers to the message pipelines. Having gotten a pointer to the message pipeline, the execution engine sends the message directly to the pipeline and returns. After that, it is up to the consummating choreography instance to fetch it from the pipeline and store it in 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.

At the end of this section, let's see how dynamic service is supported in WS-CDL/WS-CDL+. Dynamic service means that logical topology among WS-CDL/WS-CDL+ participants can be changing during the execution of choreography. WS-CDL/WS-CDL+ implements this by passing the channelType variables, or the message channels. For instance, if Server A sends the message channel to Server P and to Server B, this means B has created a message channel to P, as shown in Figure 7.

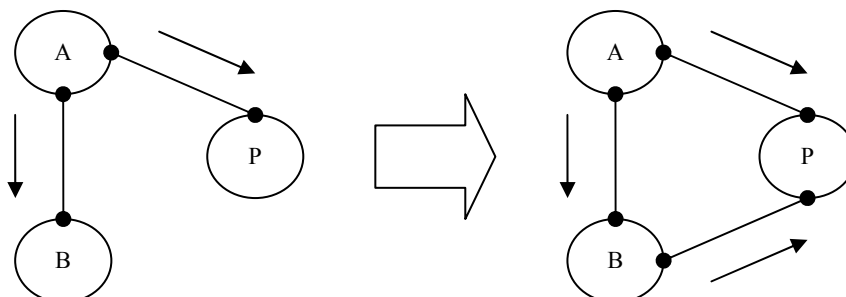


Fig. 7. An example of dynamic service

4.5 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.

The execution engine makes use of coordination messages to transfer the address table, whose definition goes like the following:

```
<coordinators instanceID="NCName" >
  <address>uri</address>+
</coordinators>
```

However, the exception itself is propagated using exception messages, which are defined like the following:

```
<exception name="xsd:QName"
  raisingRoleType="xsd:QName"?
  raisingTime="xsd:dateTime"? />
```

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 we developed for the functional test 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 8a and Figure 8b 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.

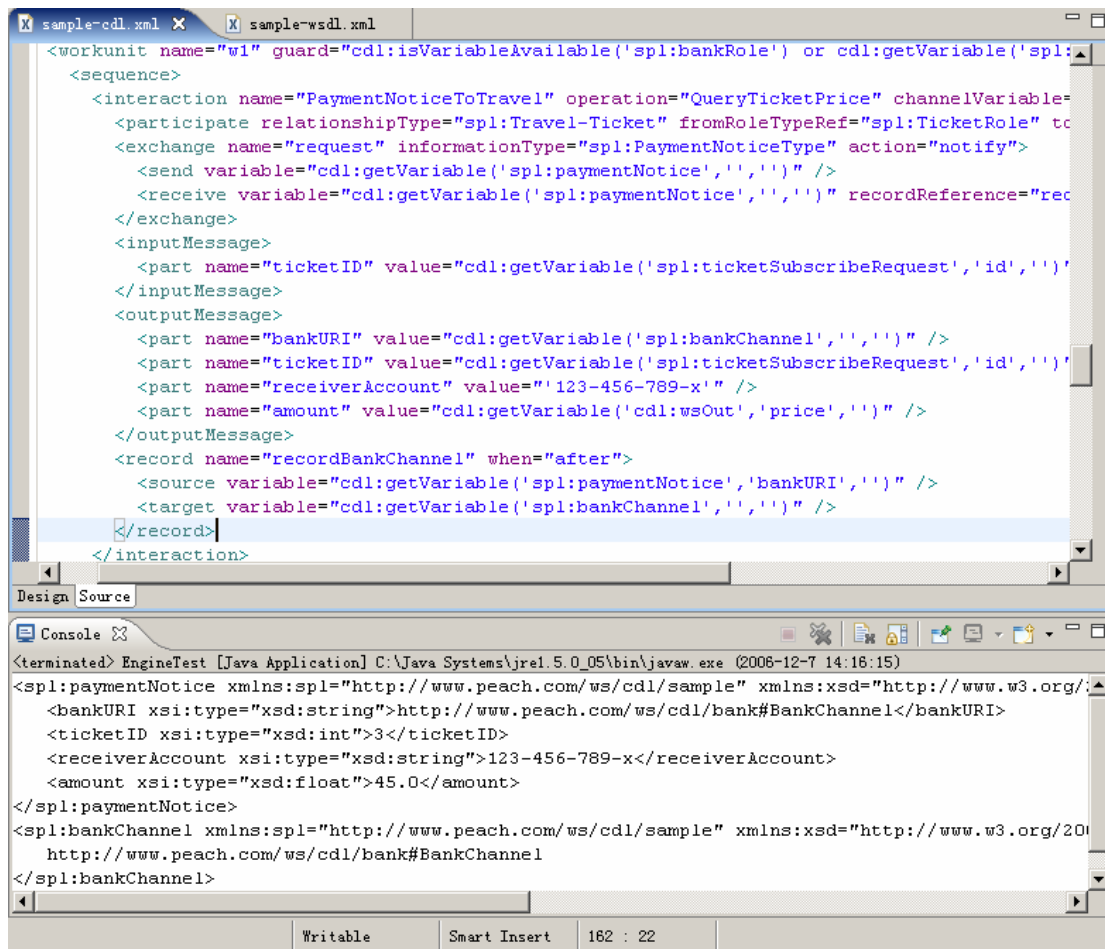


Fig. 8a. Some variable values when payment notice received from AirTicketService

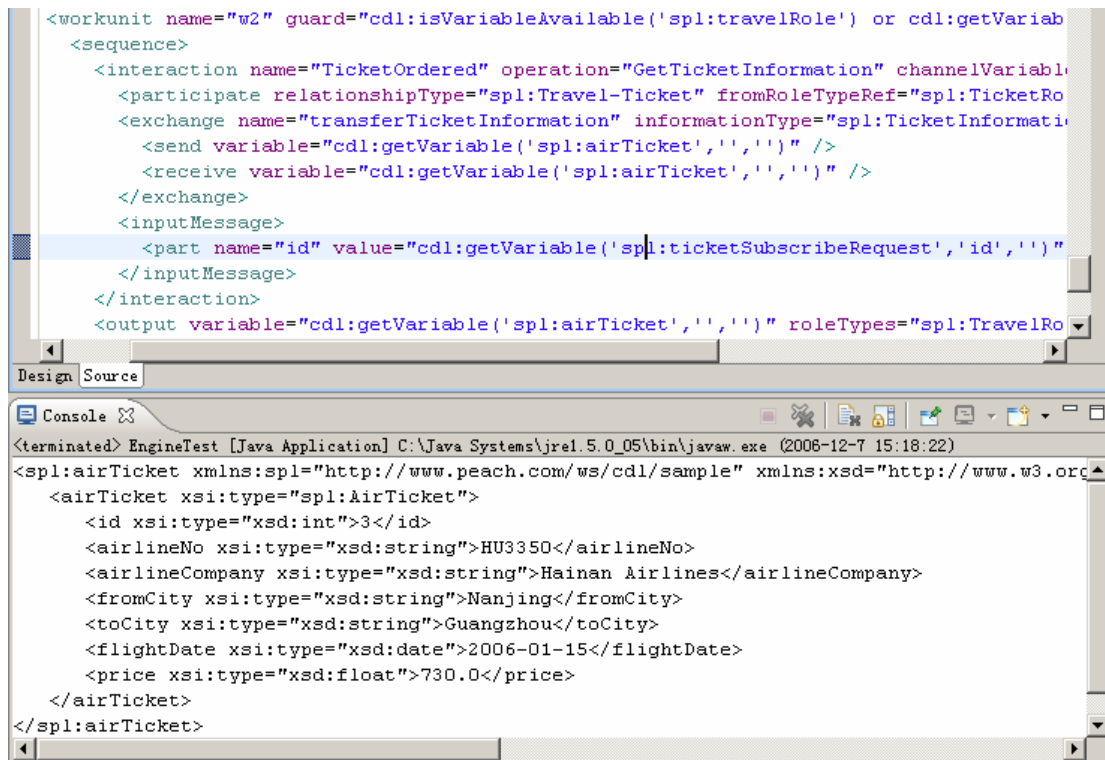


Fig. 8b. The electronic ticket received

5.2 Performance experiments

First, we will test the running speed of *interaction* activity with *request-response* and *notification* style respectively. Tests are done by repeatedly executing a specific *interaction*. The experimental data and result are respectively shown in Table 3a, 3b, 4a, and 4b (in millisecond). From these results, we conclude that the execution time for an *interaction* activity with the *request-response* style is between 6 to 8 milliseconds in our experimental environment, with the average execution time 6.64 milliseconds. The execution time for an *interaction* activity with the *notification* style is between 6 to 10 milliseconds in our experimental environment, with the average execution time 7.87 milliseconds.

TABLE 3a

Experimental data on interaction running speed with request-response style

	Loops	I	II	III	Average
①	50	921	921	906	916
②	100	1311	1327	1311	1316.3
③	150	1655	1640	1655	1650
④	200	1936	1951	1952	1946.3
⑤	250	2264	2233	2233	2243.3

TABLE 3b

Experimental result on interaction running speed with request-response style

	Time for 50 interaction	Time for one interaction
② - ①	400.3	8.01
③ - ②	333.7	6.67
④ - ③	296.3	5.93
⑤ - ④	297	5.94

TABLE 4a

Experimental data on interaction running speed with notification style

	Loops	I	II	III	Average
①	50	609	594	593	598.7
②	100	1094	1094	1078	1088.7
③	150	1484	1469	1469	1474
④	200	1828	1844	1828	1833.3
⑤	250	2171	2172	2172	2171.7

TABLE 4b

Experimental result on interaction running speed with notification style

	Time for 50 interaction	Time for one interaction
② - ①	490	9.80
③ - ②	385.3	7.71
④ - ③	359.3	7.19
⑤ - ④	338.4	6.77

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 5 (in millisecond) and Figure 9:

TABLE 5
Experimental results on running performance between Java and WS-CDL+

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

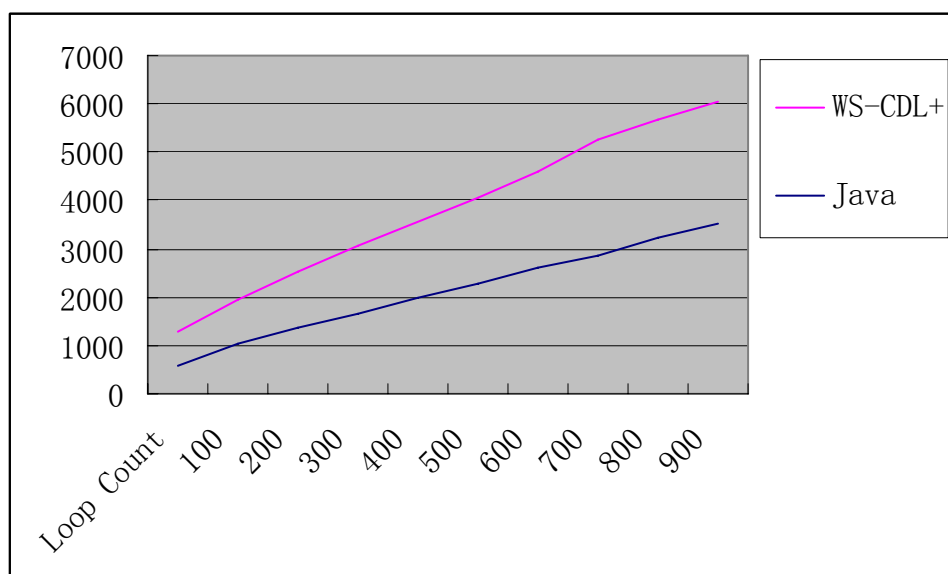


Fig. 9. The experimental results of running performance

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 it worthwhile to pay that amount of performance loss, since WS-CDL+ is able to greatly improve 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.

The methods based on Artificial Intelligence include semantic Web and AI planning, which

feature automatic or semi-automatic composition (Hao, Jinde & Jun, 2005). OWL-S and WSMO are two semantic Web methods, sponsored by W3C and OASIS respectively (Burstein, et al., 2004; “ESSI WSMO Working Group”, n.d.). In brief, either of them is an ontology of services that is used to discover, invoke, and compose Web services with a high degree of automation. OWL-S describes Web services through three main parts: the service *profile* for advertising and discovering services; the *process* model, which gives a detailed description of a service's operation via IOPE, i.e. input, output, precondition, and effect; and the *grounding*, which provides details on how to interoperate with a service, via messages (Burstein, et al., 2004). The other semantic method, WSMO, does it in a similar way. The HTN planning is one of the AI planning technologies that can be used to do service composition (Erol, Hendler & Nau, 1994). This method regards a Web service as an operator in the HTN planning, and describes its precondition, delete list, and add list. By providing the specific domain description, we can make the HTN planner work out a Web service sequence of a specific composition problem. The HTN planner includes SHOP2 and JSHOP, etc (Au, et al., 2003).

The methods based on Artificial Intelligence, no matter semantic Web or AI planning, lack of analysis and verification ability toward the composition result. Although it is proposed that OWL-S, when combined with Prolog or Petri nets, allows reasoning about correctness, the extent to which correctness is verified varies (Malek & Milanovic, 2004). Moreover, in order to reach automatic or semi-automatic composition, service providers have to give detailed semantic description to its Web service. Not only is this work undoubtedly laborious, but the correctness of the composition results is doubtful due to the inconsistent understanding of domain ontology among different service providers.

The methods based on flows mainly consist of practical technologies and theoretical models. The practical technologies include WSFL, WSCI, XLANG, BPEL4WS, WS-CDL, etc; while the theoretical models include Pi-calculus, Petri net, etc (Leymann, 2001; Arkin, et al., 2002; Thatte, 2001; Arkin, et al., 2004; Austin, et al., 2004; Burdett & Kavantas, 2004; Barreto, et al., 2005; Fletcher & Ross-Talbot, 2006; Milner, Parrow & Walker, 1992; Milner, 1993; Petri, 1962; Petri, 1979).

IBM's WSFL and BEA Systems' WSCI are the first-generation composition languages. However, because of the incompatibility between them, researchers developed the second-generation composition languages, and BPEL4WS is one of the most promising ones (Malek & Milanovic, 2004; Arkin, et al., 2004). BPEL4WS combines WSFL and WSCI with Microsoft's XLANG specification. It is supported by a great number of corporations including IBM and Microsoft, and is currently standardized by OASIS. 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 another Web service composition/collaboration description language based upon Pi-calculus (Austin, et al., 2004; Burdett & Kavantas, 2004; Barreto, et al., 2005; Fletcher & Ross-Talbot, 2006). Its 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.

Pi-calculus and Petri net belong to the theoretical models (Milner, et al., 1992; Milner, 1993; Petri, 1962; Petri, 1979). A Petri net is a directed, connected, and bipartite graph in which nodes

represent places and transitions, and tokens occupy places (Malek & Milanovic, 2004). After specifying composition with a Petri net, we can use it to prove some algebraic properties, such as absence of deadlocks or livelocks (whether composition will terminate in a finite number of steps). Pi-calculus is another theoretical model used for Web service composition. It is a calculus of communicating systems in which one can naturally express processes which have changing structure (Milner, et al., 1992). When used for Web service composition, it is able to express the changing structure of Web services, and reason about the composition's correctness using its formal toolkits. In all, both Pi-calculus and Petri net are pure theoretical models, based upon the mathematical theory, and are more concise in expression. However, their theoretical nature 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, proposed by the W3C Web Service Choreography Working Group, presents a theoretical basis of communication-centered, concurrent programming (Brown, Carbone, Honda, Milner, Ross-Talbot & Yoshida, 2006). The global calculus originates from WS-CDL, while the end-point calculus is a typed Pi-calculus. CDL is a formal model of the simplified WS-CDL, which includes important concepts related to participant roles and collaborations among them in a choreography, and models choreography with a set of participant roles and the collaboration among them (Hongli, Xiangpeng & Zongyan, 2006a; Hongli, Xiangpeng & Zongyan, 2006b). After translating WS-CDL to the input language of the model-checker SPIN, we can automatically verify the correctness of a given choreography (Holzmann, 2004; Hongli, et al., 2006b). The relationship between WS-CDL and BPEL is also a research concern of WS-CDL. Mendling and Hafner propose a translation approach and implement a transformation program, *wscdl2bpel.xslt*, capable of generating BPEL stubs from WS-CDL (Hafner & Mengling, 2005).

7 Conclusion

WS-CDL+ is based upon the current WS-CDL specification of W3C. It includes all of the features of WS-CDL and provides the extended functionalities such as user-defined functions, timers, and implicit finalization mechanism. With these extended functionalities added, developers are capable of writing more concise, readable, and maintainable choreography description documents, and easily realizing the functionalities that are hard or even impossible to implement with WS-CDL.

The prototype system of the WS-CDL+ execution engine we built is the first attempt in bringing the WS-CDL documents into execution. With this system in hand, not only can we test the features and performance of WS-CDL/WS-CDL+, but also build the WS-CDL/WS-CDL+ plug-ins for J2EE application servers such as JBoss or Tomcat with only minimal modifications.

In all, with the development of SOC and Web service, building 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. The WS-CDL specification itself will develop better, along with its execution engine or application server, finally becoming a perfect Web service collaboration technology for its users and developers.

Acknowledgements

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

References

- Apple, A.W., & MacQueen, D.B. (1991). Standard ml of new jersey. Proceedings from *International Symposium n Programming Language Implementation and Logic Programming*. New York, NY. pp 1-13.
- Arkin, A., Askary, S., Bloch, B., Curbera, F., Goland, Y., Kartha, N., et al. (Eds.). (2004). Web services business process execution language version 2.0. *Oasis*. Retrieved ..., from <http://www.oasis-open.org/committees/download.php/10347/wsbpel-specification-draft-120204.htm>
- Arkin, A., Askary, S., Fordin, S., Jekeli, W., Kawaguchi, K., Orchard, D., et al. (Eds.). (2002, August 8). Web service choreography interface (wsci) 1.0. *W3C Note*. Retrieved ..., from <http://www.w3.org/TR/wsci/>
- Au, T-C., Ilghami, O., Kuter, U., Nau, D., William, J., Wu, D., et al. (2003). SHOP2: An htn planning system. *Journal of artificial intelligence research*, 20, pp. 379-404.
- Austin, D., Barbir, A., Ferris, C., & Garg, S., (Eds.). (2004, February 11). Web services architecture requirements. In *W3C working group note*. Retrieved ..., from <http://www.w3.org/TR/wsa-reqs/>
- Austin, D., Barbir, A., Peters, E., & Ross-Talbot, S. (Eds.). (2004, March 11). Web services choreography requirements. In *W3C working draft*. Retrieved ..., from <http://www.w3.org/TR/2004/WD-ws-chor-reqs-20040311/>
- Barreto, C., Burdett, D., Fletcher, T., Kavantzias, N., Lafon, Y., & Ritzinger, G. (Eds.).

- (2005, November 9). Web services choreography description language version 1.0. In *W3C candidate recommendation*. Retrieved ..., from <http://www.w3.org/TR/2005/CR-ws-cdl-10-20051109/>
- Barros, A., Dumas, M., & Oaks, P. (2005, March). A critical overview of the web services choreography description language (WS-CDL). *BPTrends*, pp. 1-24.
- Benetallah, B., Chang, H., Dumas, M., Kalagnanam, J., Ngu, A.H.H., & Zeng, L. (2004, May). QoS-aware middleware for web services composition. *IEEE transactions on software engineering*, 30(5), pp. 311-327.
- Bouché, P. (2006). WS-CDL and pi-calculus. In F. Puhmann, H. Schuschel, & M. Weske (Eds.), *Business Process Management II – Winter Term 2005/2006* (pp. 56-71).
- Brown, A., & Haas, H. (Eds.). (2004, February 11). Web services glossary. In *W3C working group note*. Retrieved ..., from <http://www.w3.org/TR/ws-gloss/>
- Brown, G., Carbone, M., Honda, G., Milner, R., Ross-Talbot, S., & Yoshida, N. (2006, October 23). A theoretical basis of communication-centred concurrent programming. *Web services choreography working group*. Retrieved ..., from <http://www.w3.org/2002/ws/chor/edcopies/theory/note.pdf>
- Burdett, D., & Kavantzias, N. (Eds.). (2004, March 24). WS choreography model overview. In *W3C working draft*. Retrieved ..., from <http://www.w3.org/TR/2004/WD-ws-chor-model-20040324/>
- Burstein, M., Hobbs, J., Lassila, O., Martin, D., McDermott, D., McIlraith, S., et al. (2004). *OWL-S: Semantic markup for web services*. Retrieved ..., from

<http://www.daml.org/services/owl-s/1.1/overview/>

- Erol, K., Hendler, J., & Nau, D.S. (1994). HTN planning: Complexity and expressivity. Proceedind from AAAI-94: *The 12th National Conference on Artificial Intelligence*. Seattle, WA: pp. 1123-1128.
- Farong, Z., Shensheng, Z., & Yinxing, W. (2002). π -calculus for web services. Proceedings from GCC2002: *The International Workshop on grid and Cooperative Computing*. Beijing, China, pp. 800-810.
- Fletcher, T., & Ross-Talbot, S. (Eds.). (2006). Web services choreography description language: Primer. In *W3C working draft*. Retrieved ..., from <http://www.w3.org/TR/2006/WD-ws-cdl-10-primer-20060619/>
- Georgakopoulos, D., & Papazoglou, M.P. (2003). Service-oriented computing. *Communications of the ACM*, 46(10), pp. 25-65.
- Gudgin, M., Hadley, M., Karmarkar, A., Lafon, Y., Mendelsohn, N., Moreau, J.J., et al. (Eds.). (2007, April 27). SOAP version 1.2 part 1: Messaging framework (2nd ed.), *W3C recommendation*. Retrieved ..., from <http://www.w3.org/TR/soap12-part1/>
- Hafner, M., & Mendling, J. (2005). From inter-organizational workflows to process execution: Generating bpel from ws-cdl. Proceedings from OTM 2005 Workshops: *Lecture Notes in Copmuter Science 3762*. Larnaca, Cyprus: Springer-Verlag, pp 506-515.
- Hao, T., Jinde, L., & Jun, L. (2005). Describing and verifying web service using pi-calculus. *Chinese journal of computers*, 28(4), pp. 635-643.
- Holzmann, G.J. (2004). *The SPIN model checker: Primer and reference manual*.

Boston, MA: Pearson Education Inc.

Hongli, Y., Xiangpeng, Z., & Zongyan, Q. (2006a, September). A formal model for web service choreography description language (WS-CDL). Proceedings from ICWS 2006: *International Conference on Web Services 2006*. Chicago, IL.

Hongli, Y., Xizngpeng, Z., & Zongyan, Q. (2006b, September). Towards the formal model and verification of web services choreography description language. Proceedings from WS-FM '06: *3rd International Workshop on Web Services and Formal Methods*. Vienna, Austria: Springer.

Huhns, M.N., & Singh, M.P. (2005). *Service-oriented computing: Semantics, processes, agents*. West Sussex, England: John Wiley & Sons Ltd.

Kulchenko, P., Snell, J., & Tidwell, D. (2002). *Programming web services with soap*. Sebastopol, CA: O'Reilly & Associates, Inc.

Lafon, Y., & Mitra, N. (Eds.). (2007, April 27). SOAP version 1.2 part 0: Primer (2nd ed.). *W3C recommendation*. Retrieved..., from <http://www.w3.org/TR/soap12-part0/>

Lee, K., Lee, W., & Park, J. (2005, May). Incorporating preferences into web service conversations. Proceedings from WWW2005: *The 14th International World Wide Web Conference*. Chiba, Japan: Keio University.

Leymann, F. (2001). *Web services flow language (wsfl): Version 1.0*. Retrieved ..., from <http://www3.ibm.com/software/solutions/webservices/pdf/WSFL/pdf>

Limbu, D.K., Wah, L.E., & Yushi, C. (2005). Web services composition – An overview of standards. *Synthesis journal*, 4, pp. 137-150.

- Malek, M., & Milanovic, N. (2004). Current solutions for web service composition. *IEEE internet computing*, 18(6), pp. 51-59.
- Milner, R. (1993). The polyadic π -calculus: A tutorial. In F.L. Bauer, W. Bauer, & H. Schichtenberg (Eds.), *Logic and algebra of specification* (pp. 203-246). New York: Springer-Verlag.
- Milner, R., Parrow, J., & Walker, D. (1992). A calculus of mobile processes: Part I and part II. *Information and computation*, 100(1), pp. 1-77.
- Moller, F., & Victor, B. (1994). The mobility workbench: A tool for the pi-calculus. Proceedings from 6th *International Conference in Computer Aided Verification*. Stanford, CA: Springer. pp. 428-440.
- Papazoglou, M.P. (2003). Service-oriented computing: Concepts, characteristics and directions. Proceedings from WISE'03: *Fourth International Conference on Web Information Systems Engineering*. Washington, DC: IEEE.
- Petri, C.A. (1962). Kommunikation mit automaten, Ph. D. Thesis. *Institut für Instrumentelle Mathematik*. Bonn.
- Petri, C.A. (1979). Introduction to general net theory. *Lecture Notes in Computer Science*, 84, pp. 1-19.
- Ross-Talbot, S. (2005, October). Orchestration and choreography: Standards, tools and technologies for distributed workflows. Proceedings from NETTAB 2005: *The Network Tools and Applications in Biology workshops 2005*. Naples, Italy.
- Thatte, S. (2001). *XLANG: Web services for building process design*. Retrieved ..., from http://www.gotdotnet.com/team/xml_wsspecs/xlang-c/default.htm

Web Service Modeling Ontology (n.d.) *ESSI wsmo working group*. Retrieved ..., from

<http://www.wsmo.org/index.html>