International Journal on Web Service Computing (IJWSC), Vol.2, No.1, March 2011

EXTENDING WS-CDL TO SUPPORT REUSABILITY

Farhad Mardukhi, Naser NematBaksh and Kamran Zamanifar
Department of Computer Engineering, Isfahan University, Isfahan, Isfahan
{Mardukhi, Nemat, Zamanifar}@eng.ui.ac.ir

ABSTRACT

WS-CDL is a very rich language that is specially designed to describe choreography of services. However it is very poor to adopt reusability mechanisms for making the choreography easy to design and confident to use. The main challenge is that there is no mechanism to make a reusable sub choreography which is able to expose an interface. Therefore, it is impossible to inject variables like exception variables from performing choreography into performed sub choreography.

In this paper, a complex element namely Template is added to WS-CDL making it more adequate to support reusability. A template is an abstract definition of an interaction pattern which is appeared frequently through a family of business services choreographies. The paper is also details how to use the template as black box in main choreography including assigning the variables to template interface parameters. We enhanced meta model of WS-CDL by adding template related elements, then produced a simple engine that loads the our enhanced meta model of WS-CDL, the file paths of main and template choreographies and automatically generate an output file includes a compiled choreography code expressed with standard WS-CDL.

KEYWORDS

Web service, Choreography, Reusability, Interaction Pattern, Template

1. INTRODUCTION

Service-oriented architecture (SOA) is a new modern paradigm for developing distributed software systems. Each system is composed of a set of independent and interoperable entities namely Web services. *Web Services* are software components that provide self-contained functionality via Internet-enabled, interoperable interfaces for their clients in stateless communications.

A Service oriented system is usually large-scale and complex resulting in high development cost, low productivity and usually high risk to develop [2]. The most promised solution for this problem in software engineering is reusability [14]. Though, the SOA is defined on the basis of some main concepts in which reusability is one, this concept is not supported at all levels of SOA standard stack as well.

Each Web service behaves as a reusable block box entity that provides its functionality through its interface. The interface of a Web service is published as WSDL[1] [16] in UDDI[2] repository. The client invokes the computing ability of a service when it knows about its capabilities when parses WSDL. Invocation of single Web service is a simple interaction scenario however complex multi party interactions will be appeared when a numbers of services corporate to make a composite application (service). A simple interaction with a service (service invocation) actually is a type of reuse which is handled in current SOA standard stack; however the reusability of complex interaction among services is not supported [1,6]. In other words, the

[1] . Web Service Definition Language
[2] . Universal Description Discovery and Integration

functionality interface of a Web service is easily reused in current SOA standards; however the reuse of interactional (behavioral) interface of a Web service taking part in a corporation is not resolved. This paper regards to this issue in multi-party interactions among services entitled as service choreography.

The concepts of choreography come into regard when two or more parties wish to embed complex long-running multi-party interactions within their corporations [4]. Choreography defines the sequence of dependencies between interactions between multiple parties in order to implement a business corporation comprising multiple web services. W3C define ***choreography*** as a view which covers collaborative process involving services where their interactions are described from a global perspective [4,9]. From this view, WS-CDL[3] is a known and standard language supported by W3C. It is a work-flow based language which describes the interactions and their dependencies to establish choreography.

In several related researches, the reused interactions are known as interaction patterns [1,7,12]. Each interaction pattern describes a problem which occurs over and over again in collaborations, and then describes the core of the solution to that problem. Interaction patterns today play key roles for developing high quality software systems:

- Reduce a component's dependence on its environment by defining interaction protocols between components separately from the components themselves. Specify these interactions in terms of abstract interfaces, and implement components to communicate with each other through them.
- Avoid poor design of interaction protocols between collaborating components. Poor design of interactions can lead to unwanted behavior and internal dependencies that complicate system architecture. Interaction patterns
- Make the systems to be evolved accord to requirement changes.

This work is going to bring the interaction patterns into choreography of services, here WS-CDL. This goal will be met when two corresponding questions is addressed; how to define an interaction pattern and how to be reused it, both in WS-CDL syntax style.

In WS-CDL, there is a tag entitled as *Perform* used to execute a separate choreography through the main choreography. This feature only calls an independent choreography at a specified point of main choreography process. Comparing with interaction pattern concept, performed *choreography* in WS-CDL lacks capabilities expected for reusability; hence it is not true to call it as an interaction pattern. Because it does not support even the minimum requirements needed for developing and reusing interaction patterns. For example, there is no a way to parameterize the performed (reused) choreography.

2. BASIC CONCEPTS AND PROBLEM STATEMENT

Service oriented applications are usually leveraged in distributed computation environments which face several complexities like heterogeneity, dynamism and uncertainty conditions of environment and user requirements [15]. Such complexities define key requirements for service-oriented applications. The main requirement is development of independent and reusable entities which are able to interoperate and compose. The reusable elements of a service oriented application can be divided into three main categories in accordance with the architecture of SOA (figure 1):

(1) *Computation Patterns*: A computation pattern collects a set of functions delivered on its interface.

(2) *Composition Patterns*: A Composition pattern controls the computation elements to play correctly in a composition. A composition pattern is a pattern of data and control flow among computation patterns. These patterns are two main types: Workflow Patterns and

[3] . Web Service Choreography Language

Interaction Patterns [5,11]. The former describes the frequent abstract process including a sequence of tasks. The later describes an abstract interaction protocol among services.

(3) *Management Patterns*: A management pattern is usually expressed by high level pattern of rules derived from human knowledge to manage and configure the runtime behavior of components.

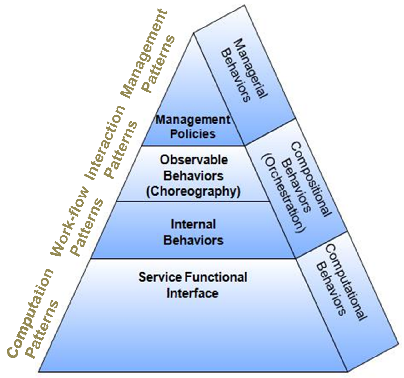


Figure 1. Pattern Types relative to SOA architecture

This work targets to develop a mechanism namely template aids us to develop and use interaction pattern in choreography of services. We point to interaction of services through business collaboration as a known term, service choreography. Choreography in view of this works, inspired from [4,9] and also the level of choreography abstraction defined in W3C specification [8], defines a set of independent roles and their interaction behaviors. Choreography is discussed from two points of views: public and local. In public perspective, choreography is a centralized abstract process basically that describes observable behaviors of services which collaborate to obtain a common goal and from local point of view, choreography is decentralized processes amongst the participants that describes interaction behavior of each service taking part in choreography. WS-CDL is a known and standard language to describe the choreography from public view and BPEL4Chor[16] and WSMO[17] are two famous that describe the choreography from local perspective.

2.1. TEMPLATES AND APPLICATIONS

Template is an abstract structure which is often used to provide a blueprint or an outline for concrete structures. In this work, a template describes an abstract choreography to explain a generic conversation among services and provides a designed solution to address similar interaction protocol appeared in a business domain. Templates are applied to various areas for developing software systems which some cases are followed.

1) **Standard business Processes**: There are many standard processes which can be used through deferent business scenarios. For example, the payment process can be used in many businesses. The payment process is developed as a template which can be adopted through consumer business processes. They have to inject associative instances into template while developing processes. Figure 2 shows this issue where a template is designed by an expert in figure 2.a and two different businesses customize it for their applications by mapping their instances (service end points, information types, exception handling logics) to template interfaces (figure 2.b and 2.c).

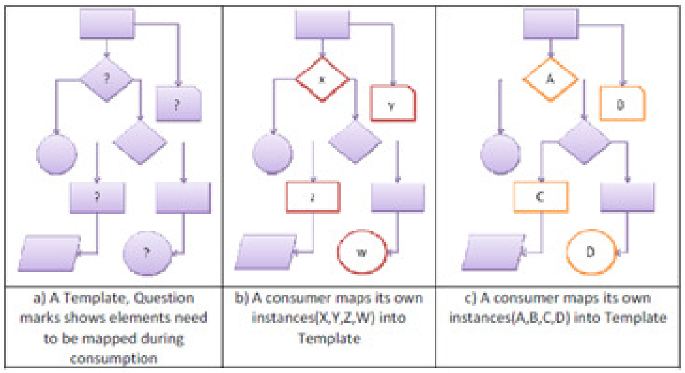

?
?
?
?
X
Y
Z
W
A
B
C
D
a) A Template, Question marks shows elements need to be mapped during consumption
b) A consumer maps its own instances(X,Y,Z,W) into Template
c) A consumer maps its own instances(A,B,C,D) into Template

Figure 2. Applying a template on two business cases

2) Implementation of Patterns: like other software architectures, SOA has its own patterns. They are classified into two categories, workflow patterns [4,8,2] and interaction patterns [5,2]. In this work, templates are specified as assistance for creating interaction patterns.
3) Discovery a specified choreography for displaying a required service behavior from a repository is an interesting application of templates. This requirement has not deal with yet in SOA. It is like service discovery, but the search items are service behavioral interface not service functionality interface. Figure 3 shows this application. Suppose a party is playing a role in choreography like CH1. He needs some services to accomplish its works. He sends its request to a directory and receives a list of candidate choreographies. After selecting more suitable one, he retrieves its role Interface from selected choreography, as a service consumer. Then he injects its own instances into retrieved Interface via a Mapping element. Actually CH1 uses CH2 as a Template. Figure 20 depicts this scenario.

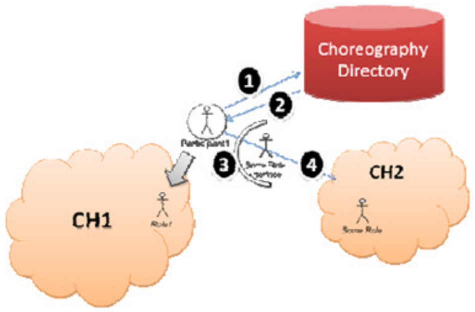


Figure 2. The repository of Service Choreography (Behaivor)

4) Templates can be used in Software Product Line (SPL) techniques to create more reusable services and functions [14]. SPL is a bunch of techniques to predict reusable parts in software. It could improve software maintenance, software quality and decrease software cost and times. Templates are very suitable facilities to implement predicted reusable parts.
5) There are a lot of unpredictable changes in service oriented applications especially within composition layer. Thus, a choreography model must be dynamic to be able to adapt itself against those changes. WS-CDL structure is not rich enough to support this requirement. The choreography is given dynamic structure when equipped by templates.

Of course, the template is not only and albeit complete way to make a dynamic choreography model.

2.2. WS-CDL AND REUSABILITY

This work focuses on WS-CDL because it is the most productive and popular language for defining the choreography and being used widely both in research and industrial projects. However, some problems are not addressed by WS-CDL as well. For example, WS-CDL has a static work-flow based structures and it is not possible to alter its parts during run time. The choreography in WS-CDL outlined in two main parts: package and choreography. The package part is a static view to choreography and declares the data types, variables, channel type, role, relationships, and so on. The process of choreography during life time is depicted at choreography part [5,9].

Also, the reusability of predefined elements of choreography is addressed very poorly in WS-CDL. This later problem is an important problem when addressed can help address first problem too[12].

Standard WS-CDL has introduced a structure namely "Perform activity" to use of external or internal sub choreographies. A main choreography calls sub choreography just as an independent choreography by performs activity. There is no ability to transfer data through the interface of sub choreography. This ability of WS-CDL is very simple and it is very far from the capabilities expected for reusability; some of weaknesses are below:

This language's documents [9] haven't said any things about reusing of some elements such as *Exception Blocks* , *Finalizer Blocks* and *Variable Definitions* within performed sub choreographies. This could be a notable weakness. Because performed choreography has to again define some static elements like *RoleTypes, Information types* and *Channel Types*, but they are package level elements and defining them through choreography part is inflexible. In this reason, sub choreographies face problems when intending to share several elements.

A possible objection to define a new extension, template, is "why have we tried to introduce a new extension instead of improving and extending *Perform* activity?" Firstly, as a general solution, the WS-CDL designers declared that defining new extensions for handling new requirements is preferable than altering standard format of its elements [9]. Secondly, on the basis of choreography definition the *Perform activity* element is used for running an instance of a called choreography at a particulate point. This is very different from those requirements the interaction patters follow. The interaction pattern is going to make the choreography easy to design. Beyond these reasons, suppose that the *Perform activity* element is extended to support reusability. In this situation, *Perform activity* must substitute all elements of performed choreography interface by corresponding elements of host choreography. When the performed choreography is defined within a package out of host choreography, the host must be aware of all shared data between its enclosed package and the package encompasses the performed choreography (figure 3).

For example CH2 which is included in a package entitled ExternalPackage uses *RoleType1*, *ChannleTypes1* and *Relationship type*. These types must be informed to host choreography, **CH1**. It means that *Relationship1* must be a connection between true roleTypes which are shared between CH1 and CH21.

This situation is against "loosely coupling" as a fundamental principle in SOA. According to this principle, *"services must be designed to interact without the need for tight and cross-service dependencies [8,12]*".

To resolve this problem we should prepare a new mechanism to inject necessary instances (e.g. *RoleTypes, Relationships* etc) into performed choreography so that performed choreographies can be designed independent of host choreographies. This new mechanism in this work is called mapping when a Template is used.

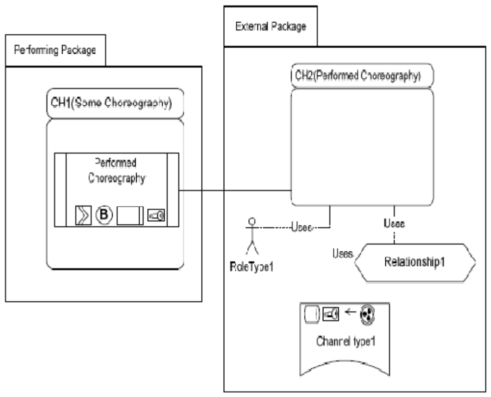


Figure 3. CH1 has to be aware of used elements by CH2

2.2. REUSABLE CHOREOGRAPHY DESIGN: OUTLINED PROCESS

In a usual development process for making a service based application, firstly the choreography process is designed, then each party of choreography must designed own behavior interface manually or automatically derived from choreography. The behavior interface of each party should be complement with internal behavior of a service to access to internal resources. Currently, the BPEL[4] is promised language to implement the behavior of a party. After generating the BPEL of each party, it is executed by orchestration engine like BizTalk, ActiveBpel etc. This works targets to bring the benefits of reusability into choreography design phase.

Reusability in choreography means developing mechanisms to design the choreography as a set choreography templates and reuse them during the design of next choreographies. Choreography template is an interaction pattern (protocol) which is appeared frequently among a family of business process.

For example, a credit card validation needs usually a common interaction scenario between the bank party and card owner. Therefore, the interactions among the card owner and bank are modeled as a template.

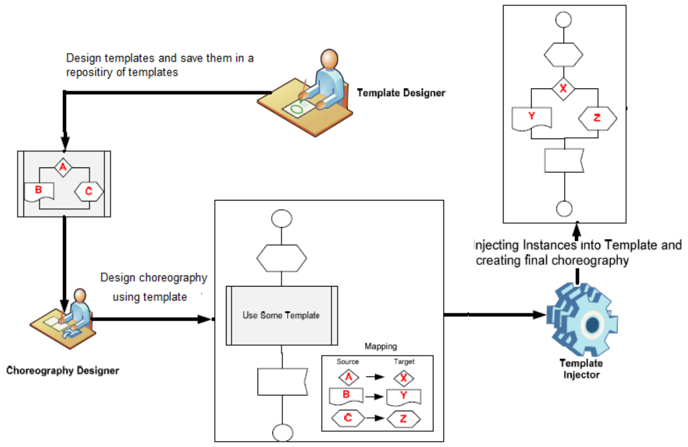

Design templates and save them in a repositiry of templates
Template Designer
A
B
C
Design choreography using template
Choreography Designer
Use Some Template
Mapping
Source
Target
A
X
B
Y
C
Z
X
Y
Z
Injecting Instances into Template and creating final choreography
Template Injector

Figure 4. The outlined process of Template-enabled Choreography design

3 TEMPLATE STRUCTURE

[4] . Business Process Execution Language

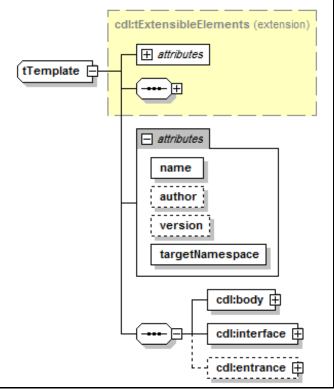
The *Template* is organized at three main parts as below. Template exposes its capabilities through *Interface*. *Entrance* is to initialize and check preconditions before Template being used, and internal contents of Template are declared within *Body* section.

$$\text{Template} = \text{Interface}^{1} + \text{Entrance}^{0..1} + \text{Body}^{1}$$

Accord to above equation, for each Template there is just one Interface and one Body and Entrance is optional part. Template is taken in host choreography by a new defined extension:*UseTemplate*. *UseTemplate* has a *Mapping* sub element to inject necessary instances from host choreography into the *Template*. This issue is called with "element injection".

3.1. TEMPLATE SYNTAX

The overall syntax for Template is shown in figure 5.a where *Name* attribute is an identifier, *Author and Version* attributes respectively define authoring properties and Versioning of the Template, and targetNamespace attribute defines address of a Namespace including all necessary data schema used for defining the Template elements. The sign "?" is attached to the parts which are optional. The figure 5.b depicts the Meta model of Template expressed by XSD[5]. XSD provides an easily approachable description of the XML Schema definition language [13].



```
<complexType name="tTemplate">
<complexContent>
<extension base="cdl:tExtensibleElements">
<sequence>
  <element name="body" type="cdl:tBody"/>
  <element name="interface" type="cdl:tInterface"/>
  <element name="entrance" type="cdl:tEntrance" minOccurs="0"/>
</sequence>
  <attribute name="name" type="NCName" use="required"/>
  <attribute name="author" type="string" use="optional"/>
  <attribute name="version" type="string" use="optional"/>
  <attribute name="targetNamespace" type="anyURI" use="required"/>
</extension>
</complexContent>
</complexType>
```

Figure 5. a) The structure of Template b) The Template Meta model

The Template is considered as a complex type of Package in choreography. Therefore, to insert Template as an extension into Package the figure 6 demonstrates the essential XSD.

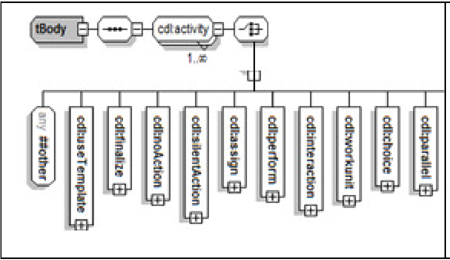
```
<complexType name="tPackage">
<complexContent>
<extension base="cdl:tExtensibleElements">
  <sequence>
    <element name="informationType" type="cdl:tInformationType" minOccurs="0"
    maxOccurs="unbounded"/>
    <element name="token" type="cdl:tToken" minOccurs="0" maxOccurs="unbounded"/>
    <element name="tokenLocator" type="cdl:tTokenLocator" minOccurs="0" maxOccurs="unbounded"/>
    <element name="roleType" type="cdl:tRoleType" minOccurs="0" maxOccurs="unbounded"/>
    <element name="relationshipType" type="cdl:tRelationshipType" minOccurs="0"
    maxOccurs="unbounded"/>
    <element name="participantType" type="cdl:tParticipantType" minOccurs="0" maxOccurs="unbounded"/>
    <element name="channelType" type="cdl:tChannelType" minOccurs="0" maxOccurs="unbounded"/>
    <element name="choreography" type="cdl:tChoreography" minOccurs="0" maxOccurs="unbounded"/>
    <element name="template" type="cdl:tTemplate" minOccurs="0 maxOccurs="unbounded"/>
  </sequence>
  <attribute name="name" type="NCName" use="required"/>
  <attribute name="author" type="string" use="optional"/>
  <attribute name="version" type="string" use="optional"/>
  <attribute name="targetNamespace" type="anyURI" use="required"/>
</extension>
</complexContent>
</complexType>
```

Figure 6. The Schema of Package element of WS-CDL when Template is added

[5] . XML Schema Definition, http://www.w3.org/TR/xmlschema11-2/

3.1. 1. Template Body Syntax

The main idea of Template extension is reusability of sub workflows. Therefore, the body section of Template should be included in dynamic structures of standard WS-CDL language [9]. They are called activity in WS-CDL. In Figure 7 we have tBody's schema in XSD.

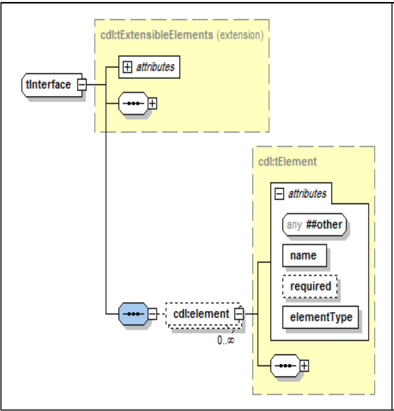


```
<complexType name="tBody">
<sequence>
<group ref="cdl:activity" maxOccurs ="unbounded"/>
</sequence>
</complexType>
```

Figure 7. The XSD schema of tBody part of Template

3.1. 3. Interfaces

The Interface is the most important concept used in Template extension. Generally, interface and inheritance are two prominent concepts for obtaining reusability especially in object oriented systems. Interfaces are used for pattern definitions, dependency injection, loosely coupling etc in programming techniques [5, 6]. An interface is observable view of an object.
A Template uses Interface to display its observable functions. Of course, the Interface in current object oriented programming has more capabilities; Nevertheless Interfaces have a sort of utilizations in this work. For example, Interface declares what elements must be mapped while element injection or what Template is appropriate for a particular purpose.
Regard to figure 8, each interface has zero or more Elements. When a Template is being used Interface's elements will be mapped to real elements of host choreography. If a template has no element to be mapped, the Interface part will be leaved empty. Each element has a unique *name* inside Template. This name is also used for mapping real instances to that element. The *elementType's* values are detailed in Figure 9 encompass the statics elements within Package and Choreography parts. TockenLocator is not included in this list, because it is not an identified element but it is a query mechanism for selecting the *Token* element. Token is a reference for a piece of data in a variable or message [4].



```
<complexType name="tInterface">
<complexContent>
<extension base="cdl:tExtensibleElements">
<sequence>
<element name="element" type="cdl:tElement" minOccurs="0"
maxOccurs="unbounded"/>
</sequence>
</extension>
</complexContent>
</complexType>
<complexType name="tElement">
<complexContent>
<extension base="cdl:tExtensibleElements">
<attribute name="name" type="NCName" use="required"/>
<attribute name="required" type="boolean" use="optional" default="true"/>
<attribute name="elementType" type="cdl:tInjectableTypes" use="required"/>
</extension>
</complexContent>
</complexType>
```

Figure 8. XSD schema for Interface

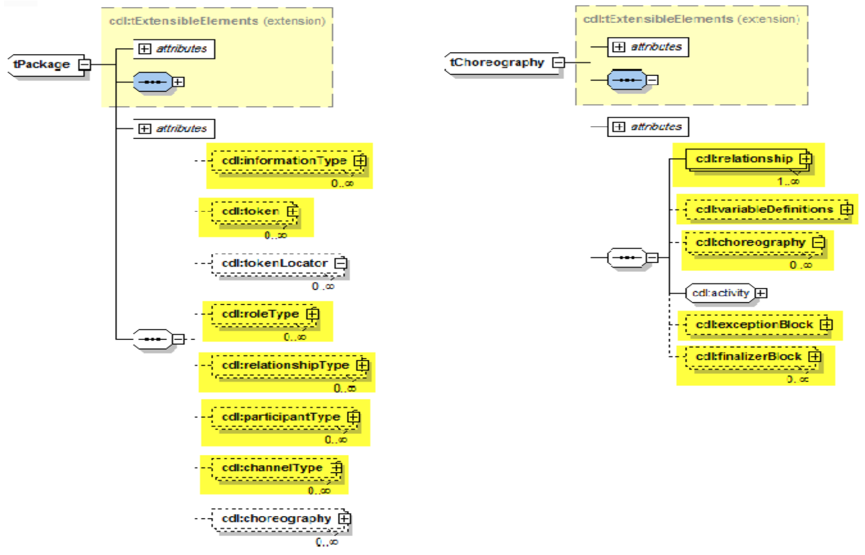


Figure 9.Acceptable elements for Package part of a Template derived from WS-CDL (The white items are not acceptable)

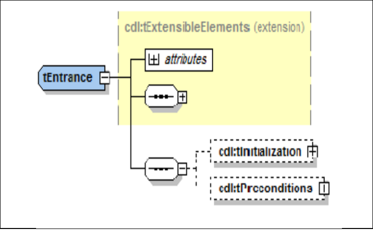
The default value for *required* attribute is *true*. When an element is required within an Interface, Template users have to map real instances of host choreography into it.

3.1.4. Entrance sub element

This element is created to initialize and check preconditions of a Template before applying. Figure 10 depicts an overview of Entrance's structure. This element accesses to all instances injected into Template, and is responsible to set up the Template. It performs two main activates: *Initialization* and *Precondition*. Firstly, Template Injector runs Initialization activity. This part helps to setting up Template injection process. After that, all conditions in Precondition part will be checked. When a condition evaluates to false, injecting process's state will be changed into Failure. Entrance is internal element for Template presents all essential information needed to initialize and deploy the Template. Figure 11 shows how designers can use Entrance element through a process namely injecting process to generate the final choreography. It includes four main steps:

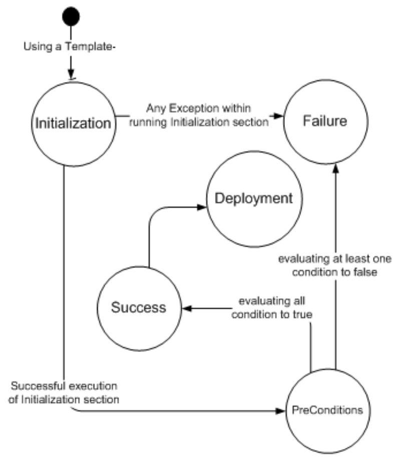
Initialization: This step provides the initial conditions for performing the injection process as detailed below:

- The mapping between the instances of host choreography to parameter of Template.
- Providing the default structures as Error exception handler.
- Providing enough capabilities to instantiate the Template and making decision about possibility of mapping all context data types (instances) to the Interface of Template.



```
<complexType name="tEntrance">
<complexContent>
<extension base="cdl:tExtensibleElements">
<sequence>
   <element name="tStandardPackage" type="cdl:tInitialization"
     minOccurs="0"/>
   <element name="tPreconditions" type="cdl:tPreconditions"
     minOccurs="0"/>
</sequence>
</extension>
</complexContent>
</complexType>
```

Figure 10. a) Outlined structure of tEntrance b)XSD schema of tEntrance element


Using a Template
Initialization
Any Exception within running Initialization section
Failure
Deployment
evaluating at least one condition to false
Success
evaluating all condition to true
Successful execution of Initialization section
PreConditions

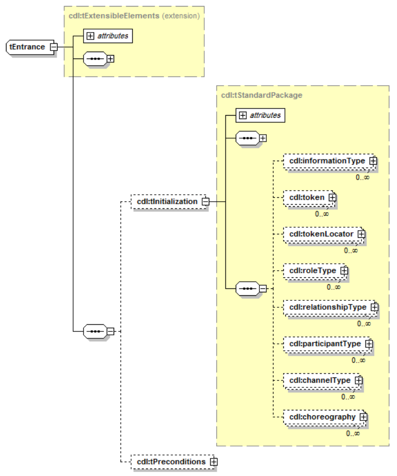
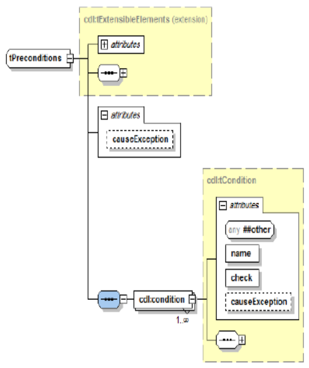


Figure 11. a) The element injection process   b) A hierarchal structure of Entrance element

Failure: is a step to handle raised errors. For example, any error exception during *Initialization* or *Precondition* parts triggers this step.
Precondition: This step checks preconditions must be met to use the Template. Each precondition is an XPath Boolean expression which declares a specified condition must be met by host choreography. For example, the requirements of security for using the Template are expressed in precondition part.



```
 <complexType name="tPreconditions">
<complexContent>
<extension base="cdl:tExtensibleElements">
<sequence>
<element name="condition" type="cdl:tCondition" maxOccurs="unbounded"/>
</sequence>
<attribute name="causeException" type="QName"use="optional"/>
</extension>
</complexContent>
</complexType>
<complexType name="tCondition">
<complexContent>
<extension base="cdl:tExtensibleElements">
<attribute name="name" type="NCName" use="required"/>
<attribute name="check" type="cdl:tBoolean-expr" use="required"/>
<attribute name="causeException" type="QName" use="optional"/>
</extension>
</complexContent>
</complexType>
```

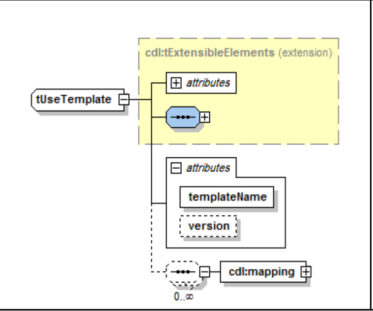
Figure 13 is the schema of tEntrance type.

Deployment: The deployment is a state which the final choreography is generated and it must be deployed on participants. The deployment can be regarded as a generating the behavioral interface of participants from generated choreography. This step is out of this paper scope.

4.5. UseTemplate element

Last sections explained how to define Template. This section presents the mechanism to use it within choreography. A new element, UseTemplate is added to WS-CDL for this purpose. It syntax is shown in figure 16. Two main arribues, *templateName* and *version*, are used to identify used Template. In spite of syntax of *Perform activity* in WS-CDL, UseTemplate doesn't have a *ChoreographyInstanceID*. Because *ChoreographyInstanceID* is a reference to an instance of choreographies are being executed during runtime. In mapping sub element, all used static structures in Template, have to be defined. They are defined in Interface part of Templates.

According to the previous sections, Template's Body consists some dynamic elements of WS-CDL. Hence, UseTemplate element just can be used in dynamic parts of a choreography definition. As a result, a new sub element is added to Activity group, as you can see in Figure 7.a.

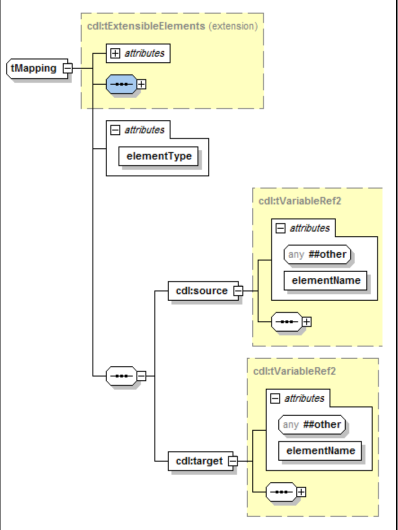


```
<complexType name="tUseTemplate">
<complexContent>
<extension base="cdl:tExtensibleElements">
<sequence minOccurs="0" maxOccurs="unbounded">
   <element name="mapping" type="cdl:tMapping"/>
</sequence>
<attribute
name="templateName" type="NCName" use="required"/>
<attribute name="version" type="string" use="optional"/>
</extension>
</complexContent>
</complexType>
```

Figure 14. The syntax of UseTemplate

4.5.1 Mapping

We have a Mapping element to inject real instances of used element into a Template. Figure 15 depicted its structure.
Target element is a reference to an element within a Template that will be bound to Source one. Then, the Source is a real instance for Target and will be injected into it. A structural exception will be raised if type of Source element doesn't mach to type of Target element, during deployment time. This type is defined in element type attribute of mapping elements.



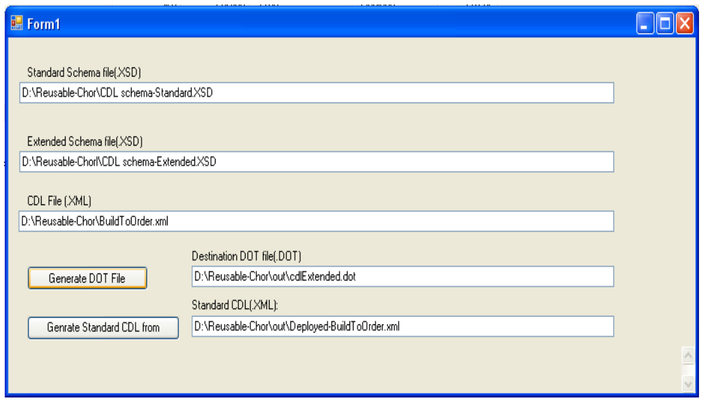
```
<complexType name="tMapping">
<complexContent>
<extension base="cdl:tExtensibleElements">
<sequence>
  <element name="source" type="cdl:tVariableRef2"/>
  <element name="target" type="cdl:tVariableRef2"/>
</sequence>
<attribute name="elementType" type="cdl:tInjectableTypes"
  use="required"/>
</extension>
</complexContent>
</complexType>
<complexType name="tVariableRef2">
<complexContent>
<extension base="cdl:tExtensibleElements">
  <attribute name="elementName" type="QName" use="required"/>
</extension>
</complexContent>
</complexType>
<simpleType name="tInjectableTypes">
<restriction base="string">
 <enumeration value="informationType"/>
  ……
 <enumeration value="finalizerBlock"/>
</restriction>
</simpleType>  </simpleType>
```

Figure 15. The XSD schema of mapping element

Target element is a reference to an element within a Template that will be bound to Source one. Then, the Source is a real instance for Target and will be injected into it. A structural exception will be raised if type of Source element doesn't mach to type of Target element, during deployment time. This type is defined in element type attribute of mapping elements.

5. IMPLEMENTATION AND CASE STUDY

For realizing the Template-enabled choreography, we built a prototype to generate the deployed choreography from host choreography that uses Template choreographies. Deployed choreography is a compiled and error-free choreography specified with standard WS-CDL. A snapshot of the prototype is shown in figure 16. There are three input files to the prototype. The first file is a standard Meta model of choreography based on WS-CDL. The second is an extended Meta model of choreography based on our extensions to WS-CDL, and third file is the source choreography which enriched by Template (figure 17). The output file of this prototype is a final choreography which is able to be deployed.



Standard Schema file(.XSD)
D:\Reusable-Chor\CDL schema-Standard.XSD

Extended Schema file(.XSD)
D:\Reusable-Chorl\CDL schema-Extended.XSD

CDL File (.XML)
D:\Reusable-Chor\BuildToOrder.xml

Destination DOT file(.DOT)
D:\Reusable-Chor\out\cdlExtended.dot

Standard CDL(.XML):
D:\Reusable-Chor\out\Deployed-BuildToOrder.xml

Figure 16. A snapshot of implemented prototype

To test the prototype, we modelled a typical *Build To Order* process by extended WS-CDL and save it in a XML file. Then, this file is given to the prototype. The prototype generated the final choreography correctly as what we had expected. The input file source code is depicted partially in figure 15.

```
<cdl:package
targetNamespace="http://www.cdlEtention/TemplateSample/"
name="BuyerSellerByTemplate" author="Farhad Mardukhi and Mahdi
Etehadi" version="1.0" xmlns:cdl="http://www.w3.org/2005/10/cdl"
xmlns:tns="http://www.cdlEtention/TemplateSample/"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="http://www.w3.org/2005/10/cdl
D:\University\MS\PAYANN~1\Final\CDLSCH~1.XSD"
xmlns:xsd="http://www.w3.org/2001/XMLSchema"
xmlns:bs="http://cdlEtention/TemplateSample/cdl/BuyerSellerExampleB
yTemplate">
<cdl:informationType name="BooleanType" type="xsd:boolean"/>
………
<cdl:informationType name="StringType" type="xsd:string"/>
<cdl:token informationType="tns:StringType" name="BuyerRef"/>
……..
<cdl:token informationType="tns:StringType" name="ShipperRef"/>
<cdl:roleType name="BuyerRoleType">
<cdl:behavior name="BuyerBehavior"/>
………
</cdl:roleType>
<cdl:relationshipType name="BuyerSeller">
…..
</cdl:relationshipType>
<cdl:relationshipType name="SellerCreditCheck">
<cdl:roleType typeRef="tns:SellerRoleType"/>
………..
</cdl:channelType>
<cdl:channelType name="Buyer2SellerChannelType">
<cdl:passing channel="tns:ToBuyerChannelType" new="true"/>
<cdl:roleType typeRef="tns:SellerRoleType"/>
<cdl:reference>
<cdl:token name="tns:SellerRef"/>
</cdl:reference>
</cdl:channelType>
…………………..
<cdl:choreography name="Main" root="true">
<cdl:relationship type="tns:BuyerSeller"/>
```

```
<cdl:template
targetNamespace="http://www.cdlEtention/TemplateSample/"
name="BarteringLoop">
<cdl:body xmlns:n1="http://www.cdlEtention/TemplateSample/">
<cdl:workunit name="BarteringLoop"
repeat="Template_barteringDone = false">
<cdl:description type="documentation">
Repeat until bartering has been completed
</cdl:description>
<cdl:choice>
<cdl:silentAction roleType="tns:SomeBuyerRoleType"/>
<cdl:sequence>
<cdl:interaction channelVariable="tns:Template_Buyer2SellerC"
name="AcceptQuoteInteraction" operation="quoteAccept">
……………………….
</cdl:interaction>
<cdl:interaction channelVariable="tns:Template_Buyer2SellerC"
name="SendChannelInteraction" operation="sendChannel">
…………………
</cdl:interaction>
<cdl:assign roleType="tns:SomeBuyerRoleType">
<………………
</cdl:assign>
</cdl:sequence>
<cdl:sequence>
<cdl:interaction channelVariable="tns:Template_Buyer2SellerC"
name="RequestNewPrice"
operation="quoteUpdate">
</cdl:interaction>
</cdl:sequence>
</cdl:choice>
</cdl:workunit>
</cdl:body>
<cdl:interface>
<cdl:element elementType="variable"
name="Template_barteringDone"/>
<cdl:element elementType="roleType"
name="Template_BuyerRoleType"/>
```

```
<cdl:relationship type="tns:SellerCreditCheck"/>
<cdl:relationship type="tns:SellerShipper"/>
<cdl:relationship type="tns:ShipperBuyer"/>
<cdl:variableDefinitions>
<cdl:variable channelType="tns:Buyer2SellerChannelType"
name="Buyer2SellerC" roleTypes="tns:BuyerRoleType
tns:SellerRoleType"/>
<cdl:sequence>
<cdl:interaction channelVariable="tns:Buyer2SellerC"
name="BuyerRequestsQuote" operation="requestForQuote">
……………………
</cdl:interaction>
<cdl:useTemplate templateName="BarteringLoop">
<cdl:mapping elementType="variable">
<cdl:source elementName="barteringDone"/>
<cdl:target elementName="Template_barteringDone"/>
</cdl:mapping>
………………………
<cdl:mapping elementType="informationType">
<cdl:source elementName="QuoteUpdateType"/>
<cdl:target elementName="Template_QuoteUpdateType"/>
</cdl:mapping>
</cdl:useTemplate>
……………………
</cdl:choice>
</cdl:sequence>
</cdl:choreography>
```

```
<cdl:element elementType="variable"
name="Template_Buyer2SellerC"/>
<cdl:element elementType="roleType"
name="Template_SellerRoleType"/>
<cdl:element elementType="relationship"
name="Template_BuyerSeller"/>
<cdl:element elementType="informationType"
name="Template_QuoteAcceptType"/>
<cdl:element elementType="channelType"
name="Template_ToBuyerChannelType"/>
<cdl:element elementType="variable"
name="Template_DeliveryDetailsC"/>
<cdl:element elementType="informationType"
name="Template_QuoteUpdateType"/>
</cdl:interface>
</cdl:template>
</cdl:package>
```

Figure 17. A typical process for *Building an Order* described by extended WS-CDL.

6. RELATED WORKS

Reusability is a major need for all software system. This is why now the component oriented development is being a promised technology in this area. Though, the most current programming language support reusability but almost able to reuse computational elements and lack reusability of interaction (composition) elements. Interaction patterns is a related idea were introduced by some researchers [1,8,5] to model the interactions among components via a set of generic interaction patterns. [18] Gathered a notable number of interaction patterns appeared at often businesses. Attempts to bring the reusability to choreography language are very low. WS-CDL as choreography standard language only supports an activity, *Perform*, to call predefined sub choreography [9]. But this feature is very far from what we expect from a real reusability mechanism.

6. CONCLUSIONS

The most promised solution for managing the complexity of modern software is reusability. Interaction pattern is an important type of reusable entities for collaborative systems. For service oriented applications, an interaction pattern is a generic conversation protocol between several services. Currently in standard service choreography (interaction) languages there is a very poor structure to support interaction patterns. For example, WS-CDL as a popular language has only a structure namely *Perform* activity to run an instance of a sub choreography at run time. This is not considered as a proper structure to bring reusability into choreography. Therefore, this work proposed a new extension, Template, to resolve this problem. The Template is a generic definition of interaction protocol between some services. The template displays its functions through its interface. The caller (host) choreography adopts the Template by mapping its elements to the parameters declared at interface of Template. This mechanism helps designers to factor the code and design the choreography easier. Also, a simple prototype was built to generate the final choreography automatically. The final choreography is a compiled and error-free choreography in WS-CDL which can be deployed to the participants of choreography. Making a repository of interaction patterns, developing the interaction pattern discovery and also making the Template more adequate are the works left for future.

REFERENCES

[1] Alistair Barros, Marlon Dumas & Arthur ter Hofstede: Service Interaction Patterns. In *Proceedings of the 3rd International Conference on Business Process Management*, Nancy, France, September 2005. Springer Verlag, pp. 302-318.

[2] M.P Papazoglou , P. Traverso, Schabram Dustdar & F. Leymann, (2007) ," Service-Oriented Computing: State of the Art and Research Challenges", Journal of Innovative Technology for Computer Professionals, IEEE Computer Society, November 2007.

[3] M. Etehadi, F. Mardukhi & N. Nematbakhsh, N, (2009). "A Graphical Representation for WS-CDL Supporting Multi Levels of Abstraction". IEEE Services Computing Conference, APSCC 2009. Asia-Pacific : 7-11 Dec. 2009.

[4] M. Barros, M. Dumas & P. Oaks. (2005),"A Critical Overview of the Web Services Choreography Description Language (WS-CDL)", BPTrends [Online accessed: Jan, 2011]. Available: http://www.bptrends.com.

[5] E. Börger & B.Thalheim, (2008), "Modeling Workflows, Interaction Patterns, Web Services and Business Processes: The ASM-Based Approach", Proceedings of the 1st international conference on Abstract State Machines, B and Z. London, UK, Springer-Verlag, p. 24-38. [Doi: 10.1007/978-3-540-87603-8_3]

[6] K. Czarnecki,(1998),"Generative Programming: Principles and Techniques of Software Engineering Based on Automated Configuration and Fragment-Based Component Models", Ph.D thesis, Technische Universität Ilmenau, Germany.

[7] E. Stroulia & J.NG, (2009),"Service interaction patterns. Proceedings of the 2009 Conference of the Center for Advanced Studies on Collaborative Research". Ontario, Canada, ACM, p. 337-337. [Doi: 10.1145/1723028.1723093]

[8] W.M. Aalst, A.J. Mooij, C. Stahl & K. Wolf, (2009),"Service Interaction: Patterns, Formalization, and Analysis. Formal Methods for Web Services", B. Marco, P. Luca Z. Gianluigi, Springer-Verlag: 42-88. [Doi: 10.1007/978-3-642-01918-0_2]

[9] W3C,"Web Services Choreography Description Language Version 1.0",(2005), W3C Candidate Recommendation, URI= http://www.w3.org/TR/2005/CR-ws-cdl-10-20051109, (online accessed Dec, 2010).

[10] W3C, "XML Path Language (XPath) Version 1.0",(1999), W3C Recommendation,URI= http://www.w3.org/TR/xpath/ (online accessed= Nov, 2010)

[11] N. Russell, A.H.M. ter Hofstede, W.M.P. van der Aalst & N. Mulyar, (2006), "Workflow Control-Flow Patterns, A Revised View", BPM Center Report BPM-06-22 , (BPMcenter.org), (online accessed= Nov, 2010).

[12] G. Decker & J. M. Zaha,(2006), "Pattern-based evaluation of WS-CDL", Queensland University, Australia, URI=http://sky.fit.qut.edu.au/~dumas/LetsDance/WSCDLEval.pdf (accessed= Jan 2011).

[13] W3C, (2005) "*WS-CDL XSD schema",* URI=http://www.w3.org/TR/2005/CR-ws-cdl-10-20051109/#WS-CDL-XSD-Schemas (online accessed = Feb 2011).

[14] F. Mardukhi & N. Nematbakhsh, (2007), " Engineering Practices and Guidelines to Enhance Reuse Adequacy in Software Product Line Analysis", SETP 2007, pp:133-140

[15] Z. Li & M. Parashar,(2006), "Enabling Dynamic Composition and Coordination for Autonomic Grid Applications using the Rudder Agent Framework", *The Knowledge Engineering Review*, Vol. 00:0, pp: 1-15

[16] G. Decker, O. Kopp, F. Leymann & M. Weske, (2007), "BPEL4Chor: Extending BPEL for Modeling Choreographies". Proceedings of the IEEE 2007 International Conference on Web Services, IEEE Computer Society, pp: 296-303.

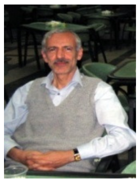
[17] D. Roman, J. Scicluna, C. Feier, M. Stollberg & D. Fensel, "Ontology-based Choreography and Orchestration of WSMO Services", http://www.wsmo.org/TR/d14/v0.1/, (online accessed= Jan 2011)

[18] N. Mulyar, W. M.P. van der Aalst1, L. Aldred & N. Russell, (2007), Service Interaction Patterns: A Configurable Framework", accessed online: http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.106.3612

Authors

F. Mardukhi received his B.Sc degree in Computer Engineering from Sharif University of Technology, Iran in 1996 and Master of Sofware Engineering from University of Isfahan, Iran in 2002. Currently he is pursuing his Ph.D degree in the Engineering Faculty og Engineering in Isfahan University. His interests include Web Service technology, Coordination problem, and adaptive software systems. He is working on dyamic and adaptive choreography models for Web services in B2B coporation.

Dr. N. Nematbakhsh received his B.Sc degree of Science in Mathamatics from Isfahan University, Iran in 1973 and Master of Science in Computer Science in 1978 from Worcester Polytechnic Institute, USA. He received the Ph.D in Computer Engineering from University of Bradford, England in 1989. He is working as Assistant Professor in the Department of Computer Engineering, Isfahan University. His expericed areas include Software Engineering Methods, Service Oriented Computig and Software Realibility.



Dr. K. Zamanifar received his B.Sc and M.Sc degree in Electrical and Electronic Engeering from University of Tehran, Iran (1976-1985). He also received the Ph.D in Computer Science from School of Computer Studies, University of Leeds, England in 1996. He is working as Associative Professor in the Department of Computer Engineering, Isfahan University. He is member of Management Committee of Computer Society of Iran and member of Iranian Association of Electrical and Electronic Enigeering. He activated at the most conferences as member of Executive Committee. His research interests are Parallel and Distributed Systems, Distributed Operating System, Concurrent Systems and Computer Supported Cooperative Work.