Enacting the Distributed Business Workflows Using BPEL4WS on the Multi-agent Platform

Li Guo, Dave Robertson, and Yun-Heh Chen-Burger

CISA, Informatics, The University of Edinburgh, United Kingdom
L.Guo@sms.ed.ac.uk, {dr, Jessicac}@inf.ed.ac.uk

Abstract. This paper describes the development of a distributed multi-agent workflow enactment mechanism using the BPEL4WS[1] specification. It demonstrates that a multi-agent protocol (Lightweight Coordination Calculus (LCC)[8]) can be used to interpret a BPEL4WS specification to enable distributed business workflow[5] using web services[2] composition on the multi-agent platform. The key difference between our system and other existing multi-agent based web services composition systems is that with our approach, a business process model(system requirement) can be adopted directly in the multi-agent system, thus reduce the effort on the validation and verification of the interaction protocol (system specification). This approach also provides us with a lightweight way of re-design of large component based systems.

1   Introduction

Composition of web services has received much interest as a means of supporting Business-To-Business or enterprise application integration. Currently, there are two main approaches for the web services composition: a static workflow technology based approach, for example, BPEL4WS, which is de facto standard for distributed workflow system using web services composition. Using such method, web services are described as activities/atomic activities in a business process model. A workflow engine is used to run the whole business process model, web services thus can be invoked as the business process executes. The basic architecture of such system is shown in figure 1. However,
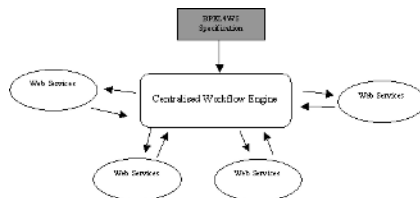

Fig. 1. The infrastructure of conventional workflow based web services composition system

the downside to this approach is that, although the workflow engine can execute these invocations asynchronously (thus generating some degree of parallelism), the process is still centralised, which means it suffers from the single point-of-failure weaknesses

that plague centralised designs[7] and in some environments, centralisation is not possible, for example, in a peer to peer mobile devices based environment. In addition, the centralised design may require heavyweight servers. Because all the interactions must go through the centralised server, if there are huge amounts of transactions taking place at the same time, the central workflow engine becomes the bottleneck of the whole system.

An alternative approach is to employ a multi-agent system for web services coordination[8,10]. With this approach, each agent $\mathcal{A}$ in the multi-agent system is associated with a web service which contains the necessary external behaviours for the participant (agent). The flow control logic is defined in the multi-agent system protocol which is passed between all the agents together with the messages to tell each agent what to do next to enable their coordination. The infrastructure of the system is depicted in figure 2. Although the centralised problem is overcome by using this ap-
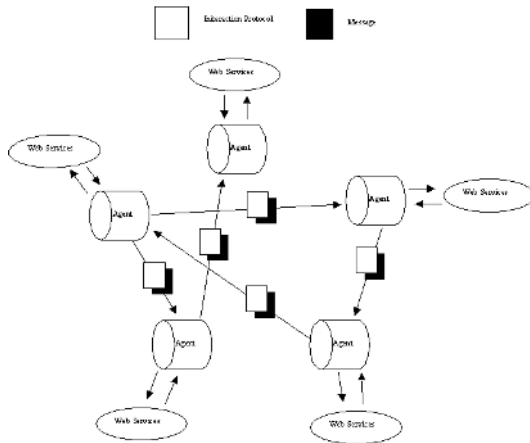

Fig. 2. The infrastructure of MAS based web services composition system

proach, a shortcoming of it is that the interaction protocol (system specification) is at a very low level of system design. It specifies the message passing that takes place between different participants at implementation level, mixing both the business and technical requirements. Therefore, huge effort on the validation and verification is required for the interaction protocol production in order to make sure that the protocol is strictly consistent with the high level requirements of the business process model.

In this paper, we propose a novel approach, with which a business process model (BPEL4WS specification) can be used to parameterise a generic multi-agent interaction protocol, thus all the existing BPEL4WS specifications and available tools can be exploited when we try to enact a distributed business workflow using web services composition on a multi-agent platform. In section 2, the necessary background introduction to the LCC protocol language and BPEL4WS is given. The infrastructure of our system is given and explained in section 3. In section 4, we explain in detail how the agents in the our system coordinate with each other using LCC protocol and BPEL4WS

specification. In section 5, we use a simple example to demonstrate how our approach works. A general discussion on our approach is given in section 6 and in section 7, the conclusion and some possible future work are addressed.

2 Background

2.1 Lightweight Coordination Calculus (LCC)

The Lightweight Coordination Calculus(LCC) is a language for representing coordination between distributed agents. In a multi-agent system the speech acts conveying information between agents are performed only by sending and receiving messages. For example, suppose a dialogue allows an agent a(r1,a1) to send a message m1 to agent a(r2,a2) and agent a(r2,a2) is expected to reply with message m2. Assuming each agent operates sequentially, the sets of possible dialogue sequences we wish to allow for the two agents in the example are as given below, where M1 $\Rightarrow$ A1 denotes a message, M1, send to A1, and M2 $\Leftarrow$ A2 denotes a message, M2, received from A2.

$$a(r1, a1) :: (m1 \Rightarrow a(r2, a2) \, then \, m2 \Leftarrow a(r2, a2))$$
$$a(r2, a2) :: (m1 \Leftarrow a(r1, a1) \, then \, m2 \Rightarrow a(r1, a1))$$

Any agent can change its role according to the definition of the dialogue:

$$a(r1, a1) :: m1 \Rightarrow a(r2, a2) \, then \, a(r3, a1)$$
$$a(r3, ID) :: m2 \Rightarrow a(r4, a3) \, then \, m3 \Leftarrow a(r4, a3)$$

The above clause means that agent a1 takes the role of r1 initially and after sending a message m1 to agent a(r2,a2), it changes its role to r3 and then takes the appropriate behaviours that are defined for a(r3,ID). This capability of LCC is very important for the our work described in this paper.

We refer to this definition of the message passing behavior of the dialogue as the *dialogue framework*. Its complete syntax can be found in [8]. A dialogue framework defines a space of possible dialogues determined by message passing, so the protocols allow constraints to be specified on the circumstances under which messages are sent or received. Two forms of constraints are permitted:

- Constraints under which message, M, is allowed to be sent to agent A. We write M $\Rightarrow$ A $\leftarrow$ C to attach a constraint C to an output message.
- Constraints under which message, M, is allowed to be received to agent A. We write M $\Leftarrow$ A $\leftarrow$ C to attach a constraint C to an input message.

For the earlier example above, to constrain agent a(r1,a1) to send message m1 to agent a(r2,a2) when condition c1 holds in a(r1,a1) we could write: m1 $\Rightarrow$ a(r2,a2) $\leftarrow$ c1.

Agent dialogue may also assume *common knowledge*, either as an inherent part of the dialogue or generated by agents in the course of a dialogue. This knowledge could be expressed in any form, as long as it can be understood by appropriate agents. We recognise the importance of preserving a shared understanding of knowledge between agents but cannot cover this issue in the current paper. As a dialogue protocol is shared

among a group of agents it is essential that each agent when presented with a message from that protocol can retrieve the *state* of the dialogue relevant to it and to that message [8].

Pulling all the above elements together, we describe a LCC dialogue protocol as the term:

$$protocol(S, F, K)$$

Where S is the dialogue state; F is the dialogue framework(sets of dialogue clauses); and K is a set of axioms defining common knowledge assumed among the agents.

To enable an distributed workflow agent to confirm a LCC protocol it is necessary to supply it with a way of unpacking any protocol it receives; finding the next moves that it is permitted to take; and updating the state of the protocol to describe the new state of dialogue. There are many ways of doing this but perhaps the most elegant way is by applying rewrite rules (more detailed re-write rules can be found in [8]) to expand the dialogues state. This works as follows[1]:

- An agent receives from some other agents a message with an attached protocol, $\mathcal{P}$, of the form $protocol(S, F, K)$. The message is added to the set of messages currently under consideration by the agent-giving the message set M_i.
- The distributed workagent extracts from $\mathcal{P}$ the dialogue clause, C_i, determining its part of the dialogue.
- Applying the rewrite rules in [8] to give an expression of C_i in terms of protocol $\mathcal{P}$ in response to the set of received messages, M_i, producing: a new dialogue clause C_n; an output message set O_n and remaining unprocess messages M_n (a subset of M_i). These are produced by applying the protocol rewrite rules exhaustively to produce the sequence:

$$\langle\ C_i \xrightarrow{M_i, M_{i+1}, \mathcal{P}, O_i} C_{i+1}, C_{i+1} \xrightarrow{M_{i+1}, M_{i+2}, \mathcal{P}, O_{i+1}} C_{i+2}, \quad ..., C_{n-1} \xrightarrow{M_{n-1}, M_n, \mathcal{P}, O_n} C_n\rangle$$

- The original clause, C_i, is then replaced in $\mathcal{P}$ by C_n to produce the new protocol, $\mathcal{P}_n$
- The distributed workflow agent can then send the messages in set O_n, each accompanied by a copy of the new protocol $\mathcal{P}_n$.

2.2   Business Process Execution Language for Web Service (BPEL4WS)

The Business Process Execution Language for Web Services (BPEL4WS) is an XML-based language for describing workflow in a distributed environment using web services. With support from IBM and Microsoft, it has become the de facto standard for workflow description. A workflow described in BPEL4WS details the flow of control and any data dependencies among a collection of web services being composed. When enacted, the composition itself becomes available as a meta-web service, eligible for inclusion in other compositions. BPEL4WS requires that all web services be described with available WSDL descriptions. The main BPEL4WS notations are given in figure 3. Due to the industry's increased focus on business process management and acceptance of BPEL4WS, vendors are producing new software tools for workflow design,

---

[1] This part is taken from the paper[8].

The Business Process Execution Language for Web Services (BPEL4WS)

A BPEL4WS workflow description is a structured XML document; as such, a collection of tags defines the BPEL4WS language's vocabulary. Here's a summary of the primary tags and their meanings:

- `<partners>` contains a list of the Web services invoked as part of this workflow;
- `<variables>` contains the variables used in this workflow;
- `<correlationSets>` provides a way to specify precedences and correlations between Web service invocations that cannot be expressed as part of the main workflow;
- `<faultHandlers>` contains exception-handling routines;
- `<compensationHandler>` handles

compensation actions if a transaction rollback occurs; and

- `<eventHandlers>` show how the workflow handles external (asynchronous) events.

Workflow logic is expressed with tags that map to traditional control flow structures:

- `<sequence>` executes the contents in a sequence,
- `<flow>` executes the contents in parallel,
- `<while>` implements a while loop,
- `<switch>` implements a case statement, and
- `<pick>` waits for external event then performs the activity associated with that event.

Within control flow structures, BPEL4WS defines tags that specify what activities to perform. These include the following:

- `<invoke>` invokes a specific Web service,
- `<receive>` receives an invocation message,
- `<reply>` sends a response message, and
- `<assign>` assigns a value, perhaps from a received message, to a variable.

The full BPEL4WS specification describes detailed semantics for the complete set of allowable tags. Additionally, the specification includes an XML Schema for BPEL4WS that can be used to validate syntactic correctness.

Fig. 3. Basic BPEL4WS Syntax[7]

specification, and enactment. An example of one such tool is IBM's BPEL4WS Java Runtime (BPWS4J) platform [6]. Think of the BPWS4J engine as an interpreter for the workflow specification: when the engine receives a workflow description, it enacts the workflow in a centralized manner.

3 A Multi-agent Platform For Distributed Business Workflow Based on BPEL4WS

A BPEL4WS specification contains all the information for running a specified business process model using web services composition, although it was not designed for decentralised multi-agent enactment and, therefore, lacks explicit instructions about how agents should coordinate. Although our multi-agent interaction protocol language (LCC) is more amendable to multi-agent enactment, it requires huge amounts of extra effort in the phases of protocol's verification and validation to ensure that the protocol is strictly consistent with the requirement. As such, the method for performing the BPEL4WS-to-multiagent-enactment is needed. The most straight forward way of doing this is to perform language mapping from BPEL4WS to LCC. Thus, any BPEL4WS specification can be translated to a LCC protocol automatically which is then used by the agents in the multi-agent system. However, an issue that we need to consider is that BPEL4WS is based on the paradigm of imperative programming langauge, while LCC is based on the declarative programming paradigm. Translating a BPEL4WS specification to a LCC protocol is actually the task of translating a imperative programme to a declarative programme, which is not possible in all circumstances.

Therefore, we choose another approach for our work: producing a LCC protocol, which acts as a BPEL4WS interpreter. The BPEL4WS specification and the LCC protocol (BPEL4WS interpreter) are passed together between the agents to enable their

coordination. This LCC protocol interpret an BPEL4WS specification so is generic for this style of process model. Based on this idea, a BPEL4WS specification that is defined in any fashion can be interpreted neatly by the LCC protocol when they are passed together in the multi-agent system. The infrastructure of the system based on this approach is given in figure 4. With this infrastructure, the multi-agent interaction protocol, the BPEL4WS specification and the messages are packed and passed together between the agents. Once an agent receives the package, it processes: the incoming message (initiating appropriate behaviors), interaction protocol and BPEL4WS (resolving the next action it needs to take), then it sends out a new package to the next agent to make the coordination continue.
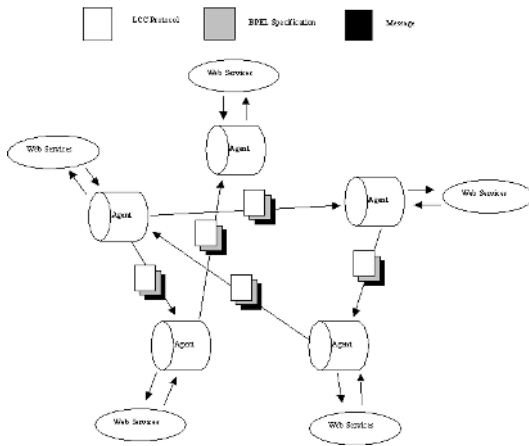
Fig. 4. The infrastructure of our generic MAS platform

4   Agent Coordination Using LCC Protocol and BPEL4WS Specification

4.1   Express BPEL4WS Specification in a Plain String Form

In order to easily interpret the BPEL4WS specification using LCC protocol, we first express the BPEL4WS specification in a plain string form rather than using its original XML syntax directly. For simplicity, only several of the main syntaxes of BPEL4WS model for our work are given below:

$$
\begin{aligned}
Model &:= \{Description, Structure\} \\
Description &:= partnerLink \left(\begin{array}{l} name(Constant), parnterLinkType(Constant), \\ myRole(Constant), partnerRole(Constant) \end{array} \right) \\
&\quad |variable(name(Constant), messageType(Constant))|... \\
Structure &:= flow([Activity/Structure, Activity/Structure, ...])| \\
&\quad switch([condition(Condition, Activity/Structure), ...])| \\
&\quad while(condition(Condition, Activity/Structure)| \\
&\quad Structure/Activity\, then\, Structure/Activity|...
\end{aligned}
$$

$$
\begin{aligned}
Activity := invoke\ &\begin{pmatrix} partnerLink(Constant), portType(Constant), \\ operation(Constant), inputVariable(Constant), \\ outputVariable(Constant), sourceLink(Constant), \\ targetLink(Constant)) \end{pmatrix} \\
|receive\ &\begin{pmatrix} partnerLink(Constant), portType(Constant), \\ operation(Constant), variable(Constant), \\ sourceLink(Constant), targetLink(Constant) \end{pmatrix} \\
|reply\ &\begin{pmatrix} partnerLink(Constant), portType(Constant), \\ operation(Constant), variable(Constant), \\ sourceLink(Constant), targetLink(Constant) \end{pmatrix} \\
|assign\ &\begin{pmatrix} from \begin{pmatrix} expression/opaque/variable(Constant), \\ property(Constant) \end{pmatrix}, \\ to(variable(Constant), property(Constant)), \\ sourceLink(Constant), targetLink(Constant) \end{pmatrix} \\
|&... \\
Condition := &Term|Condition \wedge Condition|Condition \vee Condition \\
Constant := &Term
\end{aligned}
$$

The structure (binary tree) for a BPEL4WS specification that is expressed using the above syntaxes is shown in figure 5.

4.2   Relating the Basic BPEL4WS Activities to LCC Dialogues

The only way for the agents to coordinate with each other in a multi-agent system is through message passing. Therefore, when adopting a BPEL4WS specification in a multi-agent system, the first thing we need to do is to relate the BPEL4WS syntax to message passing. Fortunately, one of the BPEL4WS design principles is to define the interaction (message passing) between two partners through centralised workflow
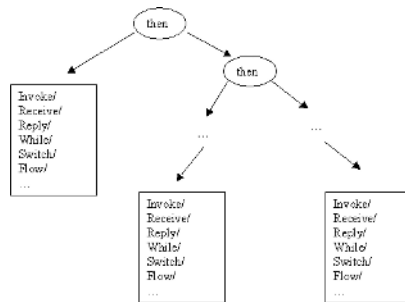

Fig. 5. The structure of the BPEL4WS model in logical form

BPEL4WS Activities	LCC Messages
<receive>	portType: operation: Variable where portType, operation, variable are extracted from the attributes of <receive>
<invoke>	portType: operation: InputVariable then portType: operation: inputVariable: outputVariable
<reply>	Variable

Fig. 6. Translations from BPEL4WS activities to LCC messages

engine. A centralised workflow engine sends and receives messages to/from the participants to enable the interaction by using some basic activities. In our system, each agent acts as a web service proxy. Instead of sending and receiving messages through a centralised server, the messages are taking place directly between participants (agents). Thus, the translation from the BPEL4WS basic activities to LCC dialogues is possible. Space limitations prevent giving the entire translation here, but a segment of it is given below:

4.3   Using LCC Protocol to Interpret the BPEL4WS Specification

In our approach, the LCC protocol is used as an interpreter to tell the agents how to process the BPEL4WS specification attached. The basic idea is: each role defined in the LCC protocol corresponds to a BPEL4WS syntax element. There are five arguments defined for each of the LCC roles:

- $Model$: is a part of BPEL4WS model that is currently processed by the LCC protocol. Because the structure of the BPEL4WS specification is a binary tree, with our approach, the deepest node is always processed first.
- $MList$: stores all the unprocessed parts of a BPEL4WS model and is used to mark the states of the BPEL4WS model's processing. Once a basic BPEL4Ws activity is reached while an agent processes the BPEL4WS model, it starts a new dialogue based on the activity and all of the unprocessed BPEL4WS model stored in $MList$ has to be passed to the next agent.
- $VList$: stores all the concrete values of the variables that are used in workflow enactment. In the centralised environment, all the information about the variables are controlled by the central server, whereas in the distributed environment, all of such information have to be passed around.
- $IDList$: is used to connect a receive activity and its corresponding reply activity.
- $Role$: represents the real participant in the interaction defined in the partnerLink.

The definitions of some of the main LCC roles are given and explained below[2]:

$$a(interpreter(Model, MList, VList, IDList, Role), A_1) ::$$
$$\left(\begin{array}{l} PortType : Operation : InputVariable \Leftarrow a(invoke(Model, MList, VList, IDList, Role_2), A_2) \\ then \\ \left(\left(\begin{array}{l} null \leftarrow Model = ..[\_, partnerLink(\_), portType(\_), operation(\_), inputVariable(\_), \\ \qquad\qquad outputVariable(null), sourceLink(\_), targetLink(\_)] \\ then \\ \left(\begin{array}{l} null \leftarrow MList = [] \\ or \\ a(interpreter(Head, Rest, VList_1, IDList, Role), A_1) \\ \leftarrow MList = [Head|Rest] \, and \, VList_1 = [InputVariable|VList] \end{array} \right) \\ or \\ \left(\begin{array}{l} PortType : Operation : InputVariable : OutputVariable \Rightarrow \\ a(invoke(Model, MList, VList, IDList, Role_2), A_2) \\ \leftarrow Model = ..[\_, partnerLink(\_), portType(\_), operation(\_), inputVariable(\_), \\ \qquad\qquad outputVariable(OutputVariable), sourceLink(\_), targetLink(\_)] \end{array} \right) \end{array} \right) \right) \end{array} \right)$$

$$\quad or \, a(sequence(Model, MList, VList, IDList, Role), A_1) \leftarrow is\_sequence(Model)$$
$$\quad or \, a(flow(Model, MList, VList, IDList, Role), A_1) \leftarrow is_f low(Model)$$
$$\quad or \, a(invoke(Model, MList, VList, IDList, Role), A_1) \leftarrow is\_invoke(Model, Role)$$
$$\quad or \, a(receive(Model, MList, VList, IDList, Role), A_1) \leftarrow is\_receive(Model, Role)$$
$$\quad or \, a(reply(Model, MList, VList, IDList, Role), A_1) \leftarrow is\_reply(Model, Role)$$
$$\quad ...$$

---

[2] The full protocol can be found at http://homepages.inf.ed.ac.uk/s0349668/Websites/tools/protocol.inst

$a(interpreter(Model, MList, VList, IDList, Role), ID)$ defined above is used to control the role's changing of the agents. Every agent takes this role first whenever it receives a message associated with the unprocessed BPEL4WS model and then changes to the appropriate role for processing the received BPEL4WS model. Only partial definitions of this role are given here for simplicity.

$$a(sequence(Model, MList, VList, IDList), A_1) ::$$
$$a(interpreter(Model_1, [Model_2|MList], VList, IDList, Role), A_1)$$
$$\leftarrow process\_sequence(Model, Model_1, Model_2)$$

$a(sequence(Model, MList, VList, IDList), A_1)$ corresponds to the BPEL4WS *sequence* activity. Once an agent takes this role, it first gets the first child element $Model_1$ of $Model$, stores the left child elements $Model_2$ in $Mlist$ and then changes its role to $interpreter$ to process $Model_1$. For the other BPEL4WS structure activities, the basic idea is same.

$$a(flow(Model, MList, VList, IDList), A_5) ::$$
$$a(interpreter(Model_1, [NewModel|MList], VList, IDList, Role), A_5)$$
$$\leftarrow process\_flow(Model, Model_1, NewModel)$$

If the role of an agent is $flow$, the agent uses the constraint $process\_flow(Model, Model_1, NewModel)$ to process the BPEL4WS flow activity ($Model$). The function of the constraint is to extract one of the child elements ($Model_1$) of $Model$ and form another flow activity ($NewModel$) using all the left child elements. An assumption we make here is the flow activity has to be processed sequentially in the distributed environment, which is the trade-off of eliminating the centralised server.

$$a(receive(Model, MList, VList, IDList, Role_1), A_1) ::$$
$$VList_2 = [Variable|VList] \text{ and } IDList_1 = [PortType : Operation : Partner : ID|IDList]$$
$$\leftarrow PortType : Operation : Variable \Leftarrow a(Partner, ID)$$
$$then$$
$$\left(\begin{array}{l} \left(\begin{array}{l} a(receive(Model, MList, VList, IDList, Role_1), A_1) \\ \quad \leftarrow \neg check\_receive(Model, PortType, Operation, Variable, Partner) \end{array} \right) \\ or \left(\begin{array}{l} a(interpreter(Head, Rest, VList_2, IDList_1, Role_2), A_1) \\ \leftarrow check\_receive(Model, PortType, Operation, Variable, Partner) \\ and\, MList = [Head|Rest] \end{array} \right) \\ or\, null \leftarrow MList = [] \end{array} \right)$$

When the role of an agent is $receive$, it waits for an incoming message and checks if this message is the appropriate one. A message is a right one if it is sent from the right *partner* of current agent and if it is defined with the right message type. If the message is not what the agent waits for, the agent keeps waiting until it receives the proper one. If the message is the right message, the agent changes its role to $interpreter$ to process the unprocessed BPEL4WS model in $MList$.

$$a(reply(Model, MList, VList, IDList, Role_1), A_1) ::$$
$$Variable1 => a(Partner, ID) \leftarrow \left(\begin{array}{l} process\_reply(Model, Partner, PortType, Operation, Variable) \\ and\, get\_ID(Partner, PortType, Operation, IDList, ID) \\ and\, look\_up(VList, Variable, Variable1) \end{array} \right)$$

An agent sends a message in reply to a message that was received from $a(Partner, ID)$. The $Partner$ and ID is stored in $IDList$ to make sure that the message is sent to the right partner.

$$a(invoke(Model, MList, VList, IDList, Role_1), A_1) ::$$
$$PortType : Operation : InputVariable \Rightarrow a(interpreter(Model, MList, VList, IDList, Role_2), A_2)$$
$$\leftarrow process\_invoke(Model, PortType, Operation, InputVariable, Role_2)$$

$$
\left(
\begin{array}{l}
then \\
\left(
\begin{array}{l}
null \leftarrow Model = ..[\_, partnerLink(\_), portType(\_), operation(\_), inputVariable(\_), \\
\qquad\qquad outputVariable(null), sourceLink(\_), targetLink(\_)] \\
or \\
\left(
\begin{array}{l}
PortType : Operation : InputVariable : OutputVariable \\
\Leftarrow a(interpreter(Model, MList, VList, IDList, Role_2), A_2) \\
then \\
\left(
\begin{array}{l}
null \leftarrow MList = [] \\
or \\
a(interpreter(Head, Rest, VList_3, IDList, Role), A_1) \\
\leftarrow MList = [Head|Rest]\ and\ VList_1 = [OutputVariable, InputVariable|VList]
\end{array}
\right)
\end{array}
\right)
\end{array}
\right)
\end{array}
\right)
$$

When an agent is of the role *invoke*, it extracts the necessary information: $PortType$, $Operation$ and $InputVariable$ from the current BPEL4WS invoke activity ($Model$) and sends it out to the next agent that is in the role of $interpreter$) for web service's invocation. If the outputVairable is defined in the current invoke activity, then there will be a response from the message receiver later on. After the sender receives the response, it will changes its role to $interpreter$ to continuously process the unprocessed BPEL4WS model.

5   A Simple Case Study

We use a simple example to illustrate how our approach works. The definition for the input BPEL4WS specification is given as follows with all the irrelevant parts ignored:

```
< process name = "loanApprovalProcess" >
  < /variables >
    < variable name = "request" messageType = "CreditInfoMessage" / >
    < variable name = "approvalInfo" messageType = "approvalMessage" / >
  < /variables >
  < partnerLinks >
    < partnerLink name = "customer" partnerLinkType = "LinkType" myRole = "approver" / >
    < partnerLink name = "approver" partnerLinkType = "LinkType" partnerRole = "approver" / >
  < /partnerLinks >
  < sequence >
    < receive name = "receive" partner = "customer" portType = "approvalPT"
              operation = "approve" variable = "request" >
    < /receive >
    < invoke name = "invokeapprover" partner = "approver" portType = "approvalPT"
             operation = "approve" inputVariable = "request" outputVariable = "approvalInfo" >
    < /invoke >
    < reply name = "reply" partner = "customer" portType = "loanApprovalPT"
             operation = "approve" variable = "approvalInfo" >
    < /reply >
  < /sequence >
< /process >
```

The basic steps for the agents in our system to coordinate using the above BPEL4WS model and LCC protocol are illustrated in figure 7 and are explained below:

- An agent, A_1, receives the BPEL4WS specification, B together with the LCC protocol, P from section 4.2. It takes the role of $a(interpreter(B, [], [], [], \_), A_1)$. It then tries the clauses that are defined in P to find the type of the B by using the constraints $is\_sequence/is\_invoke/...$) to determine the next BPEL4WS operator. For our example, the dominant operator in B is a $sequence$ activity. A_1 changes its role to $a(sequence(B, [], [], [], \_), A_1)$.
- A_1 processes B in the role of $a(sequence(B, [], [], [], \_), A_1)$ by using the constraint $process\_sequence(B, B_1, B_2)$ and gets the first element, B_1, of B and the left elements B_2 and then changes its role to $a(interpreter(B_1, [B_2], [], \_), A_1)$ to repeat the first step.
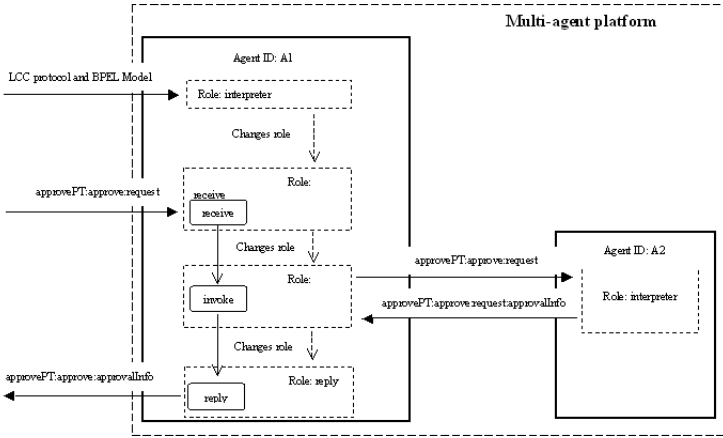
Fig. 7. Agent's coordination for performing the illustrate example

- By repeating the first step, $\mathcal{A}_1$ changes its role to $a(receive(\mathcal{B}_1, [\mathcal{B}_2], [], approver),$ $\mathcal{A}_1)$ and waits for the message $PortType : Operation : request$. Once $\mathcal{A}_1$ receives the message, following the instructions in $\mathcal{P}$, it changes its role to
  $a(interpreter(\mathcal{B}_3, [\mathcal{B}_4], [request], [PortType : Operation : Customer : CustomerID], \_), \mathcal{A}_1)$
  in which $\mathcal{B}_3$ is the first child element of $\mathcal{B}_2$ and $\mathcal{B}_4$ contains the remaining child elements of $\mathcal{B}_2$.
- By repeating the previous steps, $\mathcal{A}_1$ changes its role to $a(invoke(...), \mathcal{A}_1)$ and sends a appropriate message $\mathcal{M}$ to an agent $\mathcal{A}_2$ together with $\mathcal{P}_1$. $\mathcal{A}_2$ starts processing the $\mathcal{B}_4$ after it receives the $\mathcal{P}_1$ and $\mathcal{M}$. The coordination continues, until the processing of $\mathcal{B}$ is finished.

6 Discussion

Our approach provides an opportunity to build a multi-agent based distributed workflow system starting from a business process model rather than from a interaction protocol, which narrows the gap between the high level requirement and system specification in the development of multi-agent system and connects the business workflow community and multi-agent community. Thus, business users can produce their own business process models that can be used directly in the multi-agent system. Furthermore, since there have been many techniques and tools available for current business process modeling, they can be adopted directly for building the multi-agent system based on our approach.

Notice that the LCC protocol used to interpret BPEL4WS models is independent of any specific message passing infrastructure, although we have described it with respect to a distributed and peer to peer infrastructure, it could equally well be deployed in a more traditional server based style. Different styles of deployment are described in detail in [8]. Furthermore, the protocol can be used prior to deployment in order to predict behaviours and possible errors in interaction[10]. Another advantage is that the

workflow engine built using our approach is a real generic server. The only knowledge of it is how to process the LCC protocol and how to invoke the web services but not how to process the particular business process modelling language, which gives us a very efficient and light way for the system re-design and re-implement. Even more general, this approach can be used to adopt any functional requirement, as long as the requirement is operational and can be represented by message passing, on the multi-agent platform.

7   Conclusion and Future Work

In this paper, we have presented a novel technique for constructing distributed business workflows using existing web services composition on a generic multi-agent system platform, which particularly suits the inter-operations among enterprises. By using our approach, a BPEL4WS specification can be used directly for constructing a multi-agent system using web services composition. In such a system, all the real operations are carried by web services that are associated with distributed agents. As mentioned in the discussion section, our approach is not limited to workflow system but can fit any large component based system.

We are currently working on writing the complete LCC protocol for processing the full BPEL4WS syntaxes. We will then be able to test the protocol on a real multi-agent platform to determine various benefits and drawbacks of our approach. After this, the next stage is to solve the business level problem using our approach, such as how to do the transactional control etc.

References

1. *Business Process Execution Lanuage For Web Services specification*, http://www-128.ibm.com/developerworks/library/ws-bpel/.
2. *W3C. Web Services reference*, http://www.w3.org/2002/ws/.
3. *Web Service Definition Language references* http://www.w3.org/TR/wsdl.
4. *MagentA*, http://homepages.inf.ed.ac.uk/cdw/magenta.html.
5. *The Workflow Management Coalition*, http://www.wfmc.org/.
6. *IBM. BPWS4J*, http://www.alphaworks.ibm.com/tech/bpws4j.
7. J.M. Vidal, P. Buhler, and C. Stahl. *Multiagent systems with workflows.* IEEE Internet Computing, 8(1):76-82, January/February 2004.
8. D. Roberston, *A Lightweight Method for Coordination of Agent Oriented Web Services*, Proceedings of AAAI Spring Symposium on Sematic Web Services, 2004.
9. P. A. Buhler, J. M. Vidal, H. Verhagen *Adaptive Workflow = Web Services+Agents*, Proceeding of IEEE International Conference on Web Services 2003.
10. C. D. Walton *Model Checking Multi-Agent Web Services*, Proceeding of AAAI Symposium of Semantic Web Services 2004.
11. P. Buhler and J. M. Vidal. *Enacting BPEL4WS specified workflows with multiagent systems.* In Proceedings of the Workshop on Web Services and Agent-Based Engineering, 2004.