A Formal Model for Web Service Choreography Description Language (WS-CDL)

Yang Hongli[1], Zhao Xiangpeng[1], Qiu Zongyan[1], Pu Geguang[2], and Wang Shuling[1]

[1] LMAM and Department of Informatics, School of Math.,
Peking University, Beijing 100871, China
[2] Software Engineering Institute
East China Normal University, Shanghai, 200062, China

Abstract The Web Services Choreography Description Language (WS-CDL) is a specification of W3C developed for the description of peer-to-peer collaborations of participants from a global viewpoint. For the rigorous definition and tools support for the language, the formal semantics of WS-CDL is worth investigating. This paper proposes a small language CDL as a formal model of simplified WS-CDL, which includes many important concepts related to participant roles and collaborations among them in a choreography. The formal operational semantics of CDL is given, and static validation and verification of choreographies is studied as well. A purchase order choreography example is presented, and some properties of the proposed model are verified using the SPIN model-checker, which illustrates the potential usages and benefits of the formal model.

Keywords: Choreography, WS-CDL, Formal model, Model-checking, SPIN

1 Introduction

Web services have been becoming more and more important in the recent years, which promise the interoperability of various applications running on heterogeneous platforms. Web service composition refers to the process of combining several web services to provide a value-added service, which has received much interest to support enterprise application integration. Two levels of view to the composition of web services exist, namely orchestration and choreography. The choreography view focuses on the composition of Web services from a global perspective, and it differs from the orchestration view which focuses on the interactions among one party and others.

The recently released web service choreography description language (WS-CDL) is a W3C [1] candidate recommendation for web service composition. WS-CDL is an XML-based language for the descriptions of peer-to-peer collaborations of participants by defining, based on a global viewpoint, from their common and complementary observable behavior [13]. WS-CDL is neither an "executable business process description language" nor an implementation language. The execution logic of the application are covered by languages at another level, such as XLANG [17], WSFL [6], BPEL [2], BPML [4], etc. WS-CDL focuses on describing the business protocol among different participant roles. All the behaviors are performed by the participants, and WS-CDL gives a global observation.

Some important issues of WS-CDL are discussed in [5]. WS-CDL lacks the separation between its meta-model and its syntax, and a formal grounding. Due to the message-passing nature of web services interaction, many subtle errors can occur (e.g., message not received, deadlocks, incompatible behaviour, etc.) when a number of parties are composed together. To guarantee the correct interaction of independent, communicating web services becomes even more critical in the open-end world of web services [16]. As a language aimed to become a standard for the web service choreography, formal studies may clear the opaque points or inconsistencies in the language definition, and make the potential for the tools development.

In this paper, we propose a small language called CDL as a formal model of the simplified WS-CDL. CDL includes many important concepts related to the participant roles and the collaborations among them in a choreography. The aim of this model is to focus on the core features of WS-CDL. Based on the formal model, it is possible to reason about the properties that should be satisfied by the specified system automatically, by using existing tools, such as the model-checking tool SPIN [9], etc. We illustrate this potential by an example in this paper.

This paper is organized as follows: Section 2 is an overview of WS-CDL. We present the definition of CDL in Section 3, including its syntax and operational semantics. Section 4 gives a case study of a purchase order choreography. Some related work is discussed in Section 5, and section 6 concludes.

2 Overview of WS-CDL

This section provides an overview of the elements and structure of WS-CDL, as defined in WS-CDL specification [13] released on 9th November 2005.

A choreography defines collaborations among interacting participants. It can be recognized as a container for a collection of activities that may be performed by the participants. There are three types of activities in WS-CDL, namely control-flow activities, workunit activities and basic activities.

The control-flow activities include sequence, parallel and choice. A *sequence* activity describes one or more activities that are executed sequentially. A *parallel* activity describes one or more activities that can be executed in any order or at the same time. A *choice* activity describes the execution of one activity chosen among a set of alternative or competing activities. A *workunit* describes the conditional and repeated execution of an activity [5].

Basic activities describe the lowest level actions performed within a choreography, including:

- An *interaction* activity, which results in an exchange of information between participant roles and possible synchronization of their observable information changes and the actual values of the exchanged information.
- An *assign* activity, which assigns, within one role, the value of one variable or an expression to another variable.
- A *skip* action, which does not do anything.

In the following workunit example, the guard waits for the availability of the variable *poAck* at role *Customer*. If available, the activity will take place. Otherwise, the activity will block.

```
<workunit name="POProcess"
   guard="cdl:isVariableAvailable('poAck','','','Customer')"
   block="true">
   ... <!--some activity -->
</workunit>
```

Interaction is the most important activity in WS-CDL. An interaction activity may be composed of: (i) the participant roles involved; (ii) the exchanged information and the corresponding direction(s); (iii) the observable information changes; (iv) the operation performed by the recipient. The information exchange type of interactions is described by the possible actions on the WS-CDL channel, which falls into three types: request, respond, or request-respond. According to the exchange type, there are three kinds of interactions. The operation in an interaction activity is performed after the request (if there is one) and before the response (if there is one).

The example below shows an interaction between two roles *Consumer* and *Retailer* as a request/response exchange on the channel *retailer-channel*. The message *po* is sent from *Consumer* to *Retailer* as a request; and the message *poAck* is sent back from *Retailer* to *Consumer* as a response. After the message exchange, the variable *Consumer-poState* is assigned by the value *sent* at *Consumer*, and *Retailer-poState* by *received* at *Retailer*, as specified in the *record* elements.

```
<interaction name="createPO"
         channelVariable="retailer-channel"
         operation="handlePurchaseOrder">
 <participate
         relationshipType="tn:ConsumerRetailer
         fromRoleTypeRef="tn:Consumer"
         toRoleTypeRef="tn:Retailer"/>
 <exchange name="request"
         informationType="tn:POType"
         action="request">
  <send variable="cdl:getVariable('tn:po','','')"
         recordReference="Consumer-poState" />
  <receive variable="cdl:getVariable('tn:po','','')"
         recordReference="Retailer-poState" />
 </exchange>
 <exchange name="response"
         informationType="POAckType"
         action="respond">
  <send variable="cdl:getVariable('tn:poAck','','')"/>
  <receive variable="cdl:getVariable('tn:poAck','','')"/>
 </exchange>
 <record name="Consumer-poState" when="after">
  <source expression="sent"/>
  <target variable="cdl:getVariable('tn:poState','','')"/>
 </record>
```

```
 <record name="Retailer-poState" when="after">
  <source expression="received"/>
  <target variable="cdl:getVariable('tn:poState','','')"/>
 </record>
</interaction>
```

A role type enumerates the potential observable behaviors that a participant can exhibit in order to interact. Variables in WS-CDL are used to represent different types of information such as the exchanged information or the observable state information of the role involved. Unlike most programming languages, there is no independent variables in WS-CDL.

3 CDL: A Formal Model for WS-CDL

In this section we define a small language CDL, which can be viewed as a subset of WS-CDL. It models choreography with a set of participant roles and the collaboration among them. We give the syntax and an operational semantics here.

3.1 Syntax

In the definitions below, the meta-variable R ranges over role declarations; A and B range over activity declarations; r, f and t range over role names; x, y, u and v range over variable names; e, e_1 and e_2 ranges over XPath expressions; g, g_1, g_2 and p range over XPath boolean expressions; op ranges over the operations offered by the roles. We will use $\overline{R}$ as a shorthand for $R_1, \cdots, R_n$, for some n. (Similarly, for $\overline{x}, \overline{op}, \overline{e}$, etc.) We use $r.x$ to refer to the variable x in role r, and $r.\overline{x} := \overline{e}$ for $r.x_1 := e_1, \cdots, r.x_n := e_n$.

A choreography declaration includes a name C, some participant roles $\overline{R}$, and an activity A, with the form:

$$C[\overline{R}\ ,\ A]$$

Each participant role R has some local variables $\overline{x}$ and observable behaviors represented as a set of operations $\overline{op}$. The signature and function of the operations are defined elsewhere and omitted here. A role with name r is defined as:

$$R ::= r[\,\overline{x}\ ,\ \overline{op}\,]$$

The basic activities in CDL are the follows:

$$\begin{array}{lll} BA ::= & \mathsf{skip} & \text{(skip)} \\ & \mid r.\overline{x} := \overline{e} & \text{(assign)} \\ & \mid \mathsf{comm}\ (f.x \rightarrow t.y, rec, op) & \text{(request)} \\ & \mid \mathsf{comm}\ (f.x \leftarrow t.y, rec, op) & \text{(response)} \\ & \mid \mathsf{comm}\ (f.x \rightarrow t.y, f.u \leftarrow t.v, rec, op) & \text{(req-resp)} \end{array}$$

The skip activity does nothing. The assignment activity $r.\overline{x} := \overline{e}$ assigns, within the role r, the values of expressions $\overline{e}$ to the variables $\overline{x}$. The interaction activity is either:

- a request interaction with the form $\mathsf{comm}\ (f.x \rightarrow t.y, rec, op)$ in which the message is sent from $f.x$ to $t.y$;

– a response interaction with the form $\mathsf{comm}\,(f.x \leftarrow t.y, rec, op)$ in which the response message is sent from $t.y$ to $f.x$;
– a request-response interaction $\mathsf{comm}\,(f.x \rightarrow t.y, f.u \leftarrow t.v, rec, op)$ with a request message from $f.x$ to $t.y$ and a response message from $t.v$ to $f.u$.

In an interaction, the operation *op* specifies what the recipient should do when it receives the message. The rec is the shorthand for the assignments $f.\overline{x} := \overline{e_1}, t.\overline{y} := \overline{e_2}$, where $\overline{x}$ and $\overline{y}$ are two lists of state variables on the roles f and t respectively.

The syntax of the activities is listed here:

$$
\begin{array}{rll}
A, B ::= & BA & \text{(basic)} \\
| & p?A & \text{(condition)} \\
| & p * A & \text{(repeat)} \\
| & g : A : p & \text{(workunit)} \\
| & A; B & \text{(sequence)} \\
| & A \sqcap B & \text{(non-deterministic)} \\
| & g_1 \Rightarrow A \,[\!]\, g_2 \Rightarrow B & \text{(general-choice)} \\
| & A \parallel B & \text{(parallel)}
\end{array}
$$

An activity is either a basic activity BA, a workunit or a control-flow activity. The workunit introduced in WS-CDLis separately defined as three constructs here. Two of them are the condition construct $p?A$ and the repeat construct $p*A$, that work normally. The other is the workunit $(g : A : p)$, which will blocked until the guard g evaluates to "true". When the guard is trigged, the activity A is performed. If A terminates successfully, and if the repetition condition p evaluates to "true", the workunit will be considered again; otherwise, the workunit finishes. A control-flow activity is either a sequence activity $A; B$, a non-deterministic activity $A \sqcap B$, a general choice $g_1 \Rightarrow A \,[\!]\, g_2 \Rightarrow B$, or a parallel activity $A \parallel B$.

We introduce some well-formedness rules for CDL, which are consistent to the WS-CDL specification:

1. In a choreography, different roles have different names, and different variables in a role have different names.
2. In an interaction, each information exchange variable has the same value and same type on the sender and the receiver.
3. After an interaction, the values of state variables on the sender and the receiver are complementary to each other.
4. In a request or request-response interaction, the operation *op* should belong to the behavior interface of recipient role t; while in a response interaction, *op* should belong to role f.

In fact, the WS-CDL specification includes much more well-formedness rules. It is the responsibility of a static checker to verify the validity of all these rules. In the semantic definition, we will assume that the CDL program is well-formed.

3.2 Operational Semantics of CDL

In this section, a small-step operational semantics for CDL is presented. We define the configuration as a tuple $\langle A, \sigma \rangle$, where A is an activity, and σ is the state of the

choreography which is a composition of each participant role's state. A role state, σ_{r_i}, $i = 1, \cdots, n$, is a function from the variable names of the role r_i to their values. We suppose that each variable name is decorated with the role name on which it resides, the values of variables are unknown initially. The state of the choreography

$$\sigma \stackrel{\text{def}}{=} \langle \sigma_{r_1}, \sigma_{r_2}, \cdots, \sigma_{r_n} \rangle$$

is the composition of all the role states in the choreography.

For convenience, we use the form $\sigma[\overline{e}/r.\overline{x}]$ to denote the global state σ with some variable assignments on given role r. We use $\sigma[\overline{r.x} := \overline{e}]$ to denote the state $\sigma[\overline{e}/\overline{r.x}]$ which expresses the variable updates on one or more roles. Moreover, we use $\langle \epsilon, \sigma \rangle$ to denote the terminal configuration.

Basic Activity The semantics of the basic activities are defined as follows:

The execution of skip activity always terminates successfully, leaving everything unchanged.

$$\langle \mathsf{skip}, \sigma \rangle \longrightarrow \langle \epsilon, \sigma \rangle \tag{SKIP}$$

The $assign$ activity is a multiple assignment. The values of the variables $r.\overline{x}$ do not change until all the evaluations $\overline{e}$ are completed.

$$\langle r.\overline{x} := \overline{e},\ \sigma \rangle \longrightarrow \langle \epsilon, \sigma[\overline{e}/r.\overline{x}] \rangle \tag{ASS}$$

In an interaction activity, some information may exchange between two participant roles. After the interaction, there may be some variable updates on both roles.

$$\langle \mathsf{comm}\,(f.x \rightarrow t.y, rec, op),\ \sigma \rangle \longrightarrow \langle \epsilon, \sigma[t.y := f.x, rec] \rangle \tag{REQ}$$

$$\langle \mathsf{comm}\,(f.x \leftarrow t.y, rec, op),\ \sigma \rangle \longrightarrow \langle \epsilon, \sigma[f.x := t.y, rec] \rangle \tag{RESP}$$

$$\langle \mathsf{comm}\,(f.x \rightarrow t.y, f.u \leftarrow t.v, rec, op),\ \sigma \rangle \longrightarrow \langle \epsilon, \sigma[t.y := f.x, f.u := t.v, rec] \rangle \tag{REQ-RESP}$$

Workunit The semantics of workunit are listed as follows.

The behavior of the condition activity $(p?A)$ is the same as A when the boolean expression p evaluates to true. Otherwise, it does nothing and terminates successfully.

$$\frac{\sigma(p) = \textbf{false}}{\langle p?A, \sigma \rangle \longrightarrow \langle \epsilon, \sigma \rangle} \tag{IF-FALSE}$$

$$\frac{\sigma(p) = \textbf{true}}{\langle p?A, \sigma \rangle \longrightarrow \langle A, \sigma \rangle} \tag{IF-TRUE}$$

The repeat activity $(p*A)$ is executed by first evaluating p. When p is false, the activity terminates and nothing is changed. When p is true, the sequential composition $(A; (p*A))$ will be executed.

$$\frac{\sigma(p) = \textbf{false}}{\langle p * A, \sigma \rangle \longrightarrow \langle \epsilon, \sigma \rangle} \tag{REP-FALSE}$$

$$\frac{\sigma(p) = \textbf{true}}{\langle p * A, \sigma\rangle \longrightarrow \langle A; p * A, \sigma\rangle} \quad \text{(REP-TRUE)}$$

The workunit activity $(g : A : p)$ is blocked when the guard g evaluates to false. When g evaluates to true, A is executed. After the execution, p is tested. If p evaluates to false, then the activity terminates; if true, then the workunit restarts.

$$\frac{\sigma(g) = \textbf{true}}{\langle g : A : p, \sigma\rangle \longrightarrow \langle A; p?(g : A : p), \sigma\rangle} \quad \text{(BLOCK)}$$

Control-flow Activity The sequential composition $(A; B)$ first behaves like A; when A terminates successfully, $(A; B)$ continues by behaving like B. If A never terminates successfully, neither does $A; B$.

$$\frac{\langle A, \sigma\rangle \longrightarrow \langle A', \sigma'\rangle}{\langle A; B, \sigma\rangle \longrightarrow \langle A'; B, \sigma'\rangle} \quad \text{(SEQ)}$$

$$\langle \epsilon; B, \sigma\rangle \longrightarrow \langle B, \sigma\rangle \quad \text{(SEQ-ELIM)}$$

The non-deterministic choice $A \sqcap B$ behaves like either A or B, where the selection between them is non-deterministic, without refering the knowledge or control of the external environment.

$$\frac{\langle A, \sigma\rangle \longrightarrow \langle A', \sigma'\rangle}{\langle A \sqcap B, \sigma\rangle \longrightarrow \langle A', \sigma'\rangle} \quad \text{(NON-DET)}$$

$$\frac{\langle B, \sigma\rangle \longrightarrow \langle B', \sigma'\rangle}{\langle A \sqcap B, \sigma\rangle \longrightarrow \langle B', \sigma'\rangle} \quad \text{(NON-DET)}$$

The general choice $(g_1 \Rightarrow A \,[]\, g_2 \Rightarrow B)$ behaves like A if the guard g_1 is matched, otherwise behaves like B if g_2 is matched, where each guard is a boolean expression. If both g_1 and g_2 are matched, then the first is selected.

$$\frac{\sigma(g_1) = \textbf{true},\ \sigma(g_2) = \textbf{false}}{\langle g_1 \Rightarrow A \,[]\, g_2 \Rightarrow B, \sigma\rangle \longrightarrow \langle A, \sigma\rangle} \quad \text{(CHOICE)}$$

$$\frac{\sigma(g_1) = \textbf{false},\ \sigma(g_2) = \textbf{true}}{\langle g_1 \Rightarrow A \,[]\, g_2 \Rightarrow B, \sigma\rangle \longrightarrow \langle B, \sigma\rangle} \quad \text{(CHOICE)}$$

$$\frac{\sigma(g_1) = \textbf{true}, \sigma(g_2) = \textbf{true}}{\langle g_1 \Rightarrow A \,[]\, g_2 \Rightarrow B, \sigma\rangle \longrightarrow \langle A, \sigma\rangle} \quad \text{(CHOICE)}$$

We use interleaving semantics for the parallel composition:

$$\frac{\langle A, \sigma\rangle \longrightarrow \langle A', \sigma'\rangle}{\langle A \parallel B, \sigma\rangle \longrightarrow \langle A' \parallel B, \sigma'\rangle} \quad \text{(PARA)}$$

$$\frac{\langle B, \sigma\rangle \longrightarrow \langle B', \sigma'\rangle}{\langle A \parallel B, \sigma\rangle \longrightarrow \langle A \parallel B', \sigma'\rangle} \quad \text{(PARA)}$$

$$\langle \epsilon \parallel B, \sigma\rangle \longrightarrow \langle B, \sigma\rangle \quad \text{(PARA-ELIM)}$$

$$\langle A \parallel \epsilon, \sigma\rangle \longrightarrow \langle A, \sigma\rangle \quad \text{(PARA-ELIM)}$$

3.3 Reasoning Based on Semantics

As a simple example to show how CDL and its semantics can help to reason on choreography issues, We first illustrate the execution of an activity $poRequest; creditCheck \parallel invCheck$, which is defined in Section 4.

(1) $\langle poRequest, \sigma\rangle \rightarrow$
$\langle \epsilon, \sigma[Buyer.po/Seller.po, Seller.poAck/Buyer.poAck,$
$\text{“}sent\text{”}/Buyer.poState, \text{“}received\text{”}/Seller.poState]\rangle$ (REQ-RESP)

(2) $\langle creditCheck, \sigma\rangle \rightarrow$
$\langle \epsilon, \sigma[Credit.ccReq/Seller.ccReq, Seller.ccResp/Credit.ccResp,$
$\text{“}sent\text{”}/Credit.ccState, \text{“}received\text{”}/Seller.ccState]\rangle$ (REQ-RESP)

(3) $\langle invCheck, \sigma\rangle \rightarrow$
$\langle \epsilon, \sigma[Inv.icReq/Seller.icReq, Seller.icResp/Inv.ccResp,$
$\text{“}sent\text{”}/Inv.icState, \text{“}received\text{”}/Seller.icState]\rangle$ (REQ-RESP)

(4) $\langle \epsilon \parallel invCheck, \sigma\rangle \rightarrow$
$\langle \epsilon, \sigma[Inv.icReq/Seller.icReq, Seller.icResp/Inv.ccResp,$
$\text{“}sent\text{”}/Inv.icState, \text{“}received\text{”}/Seller.icState]\rangle$ (3, PARA-ELIM)

(5) $\langle creditCheck \parallel invCheck, \sigma\rangle \rightarrow$
$\langle \epsilon, \sigma[Credit.ccReq/Seller.ccReq, Seller.ccResp/Credit.ccResp,$
$Inv.icReq/Seller.icReq, Seller.icResp/Inv.ccResp,$
$\text{“}sent\text{”}/Credit.ccState, \text{“}received\text{”}/Seller.ccState,$
$\text{“}sent\text{”}/Inv.icState, \text{“}received\text{”}/Seller.icState]\rangle$ (2, 4, PARA)

(6) $\langle \epsilon; creditCheck \parallel invCheck, \sigma\rangle \rightarrow$
$\langle \epsilon, \sigma[\cdots, \text{“}sent\text{”}/Credit.ccState, \text{“}received\text{”}/Seller.ccState,$
$\text{“}sent\text{”}/Inv.icState, \text{“}received\text{”}/Seller.icState]$ (5, SEQ-ELIM)

(7) $\langle poRequest; creditCheck \parallel invCheck, \sigma\rangle \rightarrow$
$\langle \epsilon, \sigma[\cdots, \text{“}sent\text{”}/Buyer.poState, \text{“}received\text{”}/Seller.poState,$
$\text{“}sent\text{”}/Credit.ccState, \text{“}received\text{”}/Seller.ccState,$
$\text{“}sent\text{”}/Inv.icState, \text{“}received\text{”}/Seller.icState]\rangle$ (1, 6, SEQ)

Then we show the execution of activity $poRespond$, which selects one activity from $poResponse$ and $poReject$ based on guard condition.

(1) $\langle poResponse, \sigma\rangle \rightarrow$
$\langle \epsilon, \sigma[Buyer.poResp/Seller.poResp, \text{“}complete\text{”}/Buyer.poState,$
$\text{“}complete\text{”}/Seller.poState]\rangle$ (RESP)

(2) $\langle poReject, \sigma\rangle \rightarrow$
$\langle \epsilon, \sigma[Buyer.poRej/Seller.poRej, \text{“}complete\text{”}/Buyer.poState,$
$\text{“}complete\text{”}/Seller.poState]\rangle$ (RESP)

(3) $\langle poRespond, \sigma\rangle \rightarrow$
$\langle \epsilon, \sigma[\cdots, \text{“}complete\text{”}/Buyer.poState, \text{“}complete\text{”}/Seller.poState]\rangle$
(1, 2, CHOICE)

From step (3), after performing the activity $poRespond$, the σ will reach the state in which $poState$ has value "$complete$" on both roles $Buyer$ and $Seller$.

Combining the two parts together and applying the rule (SEQ), we have that

$$\langle purchaseOrder, \sigma\rangle \rightarrow^*$$
$$\langle \epsilon, \sigma[\cdots, \text{“}complete\text{”}/Buyer.poState, \text{“}complete\text{”}/Seller.poState,$$
$$\text{“}sent\text{”}/Credit.ccState, \text{“}received\text{”}/Seller.ccState,$$
$$\text{“}sent\text{”}/Inv.icState, \text{“}received\text{”}/Seller.icState]\rangle$$

Please note that σ will always reach the above state, no matter which rule we choose during the parallel execution. This can be viewed as a proof of the fact that "the choreography will always terminate and, when it terminates, $poState$ will always has value "$complete$".

4 Case Study: Purchase Order Service Choreography

In this section, we develop a purchase order service choreography example [14], using CDL. With the help of the semantics of CDL, we can translate a system specification in CDL rigorously into other notations. Here we show how to translate the example into the notation of SPIN verifier and then verify the properties automatically using SPIN. An automatic translator is under development.

4.1 The Example Description

This multi-participants choreography involves four participant roles: Buyer, Seller, Credit Checking Service and Inventory Checking Service. The Buyer initiates an interaction with the Seller by requesting the Purchase Order. After receiving the request, the Seller acknowledges the receipt of the Purchase Order with the Buyer, and initiates two interactions:

- Check Buyer's credit with the Credit Checking Service;
- Check product availability with the Inventory Service

The Seller's final response to the Buyer is decided by the information it receives from the two checking services:

- If both interaction results are positive, the order is approved and the Purchase Order Response message is sent to the Buyer;
- If any of the two interactions indicates a negative result, a Purchase Order Rejection message is sent to the Buyer

Figure 1 shows the control flow of the interactions, while Figure 2 illustrates the communications between the roles as a UML collaboration diagram. The number 1.1.1, 1.1.2, etc. denotes the sequence of the messages. Figure 2 should be viewed as one of the two possible instances of the interleaved execution process, where $creditCheck$ is executed first and then $invCheck$ is executed.

4.2 CDL Specification

The purchase order choreography *PurchaseOrderChor* includes a set of roles ($buyer$, $seller$, $credit$, inv) and an activity $purchaseOrder$:

$$\textit{PurchaseOrderChor}[(buyer, seller, credit, inv), purchaseOrder]$$

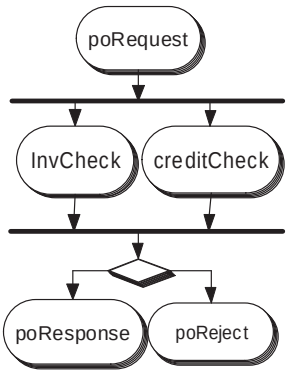

Figure 1. A Simplified View of the Purchase Order Service

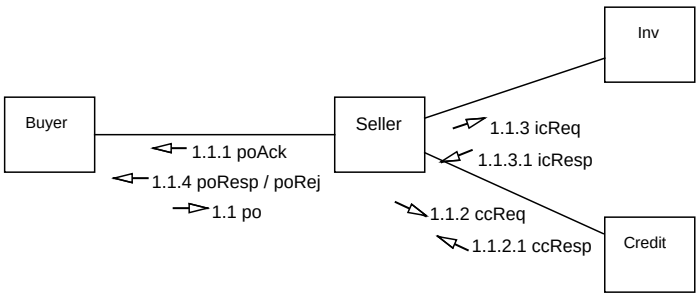

Figure 2. Communications in Purchase Order Service

The role *buyer* has name *Buyer* and a set of observable variables: purchase order *po*, purchase order acknowledgement *poAck*, purchase order response *poResp*, purchase order rejection *poRej* and purchase order state *poState*. It also has two observable behaviors: *PoReqOP* and *DisplayOP*.

$$buyer = \textit{Buyer}[(\textit{po, poAck, poResp, poRej, poState}),\ \textit{(PoReqOP, DisplayOP)}]$$

The role *seller* has name *Seller* and a set of observable variables: purchase order *po*, purchase order acknowledgement *poAck*, purchase order response *poResp*, purchase order reject *poRej*, purchase order state *poState*, credit check request *ccReq*, credit check response *ccResp*, credit check state *ccState*, inventory check request *icReq*, inventory check response *icResp*, inventory check state *icState*; and observable behavior *PoHandleOP*.

$$seller = \textit{Seller}[(\textit{po, poAck, poResp, poRej, ccReq, ccResp,}\ \textit{icReq, icResp, poState, ccState, icState}), \textit{PoHandleOP}]$$

The role *credit* has name *Credit*, a set of observable variables: credit check request *ccReq*, credit check response *ccResp*, credit check state *ccState*; and observable behavior *CreditCheckOP*.

$$credit = \textit{Credit}[(\textit{ccReq, ccResp, ccState}), \textit{CreditCheckOP}]$$

The role *inv* has name *Inv*, a set of observable variables: inventory check request *icReq*, inventory check response *icResp*, inventory check state *icState*; and observable behavior *InvCheckOP*.

$$inv = \mathit{Inv}[(\mathit{icReq},\ \mathit{icResp},\ \mathit{icState}), \mathit{InvCheckOP}]$$

The interaction activity $poRequest$ has two participant roles *Buyer* and *Seller*, a purchase order request exchange *po*, and a purchase order acknowledgement exchange *poAck*. After the interaction, the state variable *poState* in role *Buyer* is set to "sent", while in role *Seller* it is "received".

$$\begin{aligned} poRequest = \mathsf{comm}\ (&\mathit{Buyer.po} \rightarrow \mathit{Seller.po},\\ &\mathit{Buyer.poAck} \leftarrow \mathit{Seller.poAck},\\ &\mathit{Buyer.poState} := \mathit{"sent"},\\ &\mathit{Seller.poState} := \mathit{"received"}, \mathit{PoHandleOP}) \end{aligned}$$

The interaction activity $creditCheck$ has two participant roles *Seller* and *Credit*, a credit checking request exchange *ccReq*, and a credit check response exchange *ccResp*. After the interaction, the state variable *ccState* in role *Seller* is "received", in role *Credit* is "sent".

$$\begin{aligned} creditCheck = \mathsf{comm}\ (&\mathit{Seller.ccReq} \rightarrow \mathit{Credit.ccReq},\\ &\mathit{Seller.ccResp} \leftarrow \mathit{Credit.ccResp},\\ &\mathit{Credit.ccState} := \mathit{"sent"},\\ &\mathit{Seller.ccState} := \mathit{"received"}, \mathit{CreditCheckOP}) \end{aligned}$$

The interaction activity $InvCheck$ has two participant roles *Seller* and *Inv*, a invertory checking request exchange *icReq*, and a invertory check response exchange *icResp*. after the interaction, the state variable *icState* in role *Seller* is "received", in role *Inv* is "sent".

$$\begin{aligned} invCheck = \mathsf{comm}\ (&\mathit{Seller.icReq} \rightarrow \mathit{Inv.icReq},\\ &\mathit{Seller.icResp} \leftarrow \mathit{Inv.icResp},\\ &\mathit{Inv.icState} := \mathit{"sent"},\\ &\mathit{Seller.icState} := \mathit{"received"}, \mathit{InvCheckOP}) \end{aligned}$$

The interaction activity $poResponse$ has two participant roles $Buyer$ and *Seller*. The response message *poResp* is sent from role *Seller* to role *Buyer*.

$$\begin{aligned} poResponse = \mathsf{comm}\ (&\mathit{Buyer.poResp} \leftarrow \mathit{Seller.poResp},\\ &\mathit{Buyer.poState} := \mathit{"complete"},\\ &\mathit{Seller.poState} := \mathit{"complete"}, \mathit{DisplayOP}) \end{aligned}$$

In the interaction activity $poReject$, the response message *poRej* is sent from role *Seller* to role *Buyer*.

$$\begin{aligned} poReject = \mathsf{comm}\ (&\mathit{Buyer.poRej} \leftarrow \mathit{Seller.poRej},\\ &\mathit{Buyer.poState} := \mathit{"complete"},\\ &\mathit{Seller.poState} := \mathit{"complete"}, \mathit{DisplayOP}) \end{aligned}$$

The activity $poRespond$ is defined as: if both variables *Seller.icResp* and *Seller.ccResp* are "ok", then interaction activity *poResponse* is happened, otherwise, interaction activity *poReject* is happened.

$$\begin{aligned} poRespond = &(\mathit{Seller.ccResp}=\mathit{"Ok"} \wedge \mathit{Seller.icResp}=\mathit{"Ok"}) \Rightarrow poResponse\\ &[]\\ &(\mathit{Seller.ccResp}=\mathit{"notOk"} \vee \mathit{Seller.icResp}=\mathit{"notOk"}) \Rightarrow poReject \end{aligned}$$

The main activity $purchaseOrder$ performed by choreography *PurchaseOrderChor* is defined as:

$$purchaseOrder = poRequest;\ (creditCheck \parallel invCheck);\ poRespond$$

4.3 Verification using SPIN

With the semantics given above, we have modeled the purchase order service example using the SPIN model-checker. Each role is modeled by a proctype, which is a process running in parallel with other processes. The interaction is modeled by message channels and global variables. The following is a code snippet of the translated choreography, which describes the Buyer role:

```
active proctype buyer() {
  buyer_to_seller ! po;
  buyer_poState = sent;
  seller_to_buyer ? poAck;
  if
    :: seller_to_buyer ? poRej ->
       buyer_poState = complete
    :: seller_to_buyer ? poResp ->
       buyer_poState = complete
  fi
}
```

To model parallel composition, we use two auxiliary proctypes `doIC` and `doCC`, together with two auxiliary variables `finished_doIC` and `finished_doCC`, as shown below in the Seller role:

```
active proctype seller() {
  ...
  run doIC(); run doCC();
  /*inventory check & credit check */
  if
    :: para_aux ? finished_doIC ->
       para_aux ? finished_doCC
    :: para_aux ? finished_doCC ->
       para_aux ? finished_doIC
  fi;  /* wait until both processes are finished */
  ...
}
proctype doCC() {
  seller_to_credit ! ccReq;
  credit_to_seller ? ccResp(seller_ccResp);
  seller_ ccState = received;
  /* indicate that this process has finished */
  para_aux ! finished_doCC;
}
```

The rest parts of the code are omitted here due to space limitation. A full version can be found in the appendix.

Using LTL (Linear Temporal Logic), we can automatically verify many properties of the choreography, such as:

- the system will never deadlock (`timeout` is a reserved word in Promela, which is the modeling language of SPIN.)

  ```
  [] (!timeout)
  ```

– the buyer will eventually reach the "complete" purchase order state (the system will never livelock)

```
<> (buyer_poState == complete)
```

– the purchase order will be either accepted or rejected, but not both accepted and rejected (functional requirement)

```
<> (msg==poRej || msg==poResp)
```

```
! ( <> (msg==poRej) && <> (msg==poResp))
```

where auxiliary variable msg records the final message sent to the buyer.

According to our test, the verification procedure only costs several seconds on a Pentium IV machine with 512MB memory. It is possible to implement an automatic translator from CDL specification to Promela code, which is our on-going work.

5 Related Work

Formal approaches are useful in analyzing and verifying web service properties. There are some existing work on specifying and verifying web service compositions. H.Foster [8] discussed a model-based approach to verify web service compositions. G.Salaun et al. developed a process algebra to derive the interactive behavior of a business process out from a BPEL specification [16], while A.Brogi et al. presented the formalization of Web Service Choreography Interface (WSCI) using a process algebra approach(CCS), and discussed the benefits of such formalization [3]. There are also work on the formal semantics of web services languages. In the previous work, we presented an operational semantics to a simplified version of BPEL with some important concepts related to fault and compensation handling [7,18].

In a recent paper [11], N.Busi et al. proposed a simple choreography language whose main concepts are based on WS-CDL. Different from our language, this language splited the request and response message exchange in one interaction, and there was no state record variables. Also, it didn't consider the verification of web service properties. Other work includes [10], in which Misra proposed a new programming model for the orchestration of web services. It was quite far from practice and need further investigation.

The choreography working group of W3C has also recognized the importance of providing a formal grounding for WS-CDL language. Although WS-CDL appears to borrow terminologies from Pi-Calculus, the link to this or any other formalism is not clearly established [12,15].

6 Conclusion and Future Work

The goal of the WS-CDL language is to propose a declarative, XML based language that concerns about global, multi-party, peer-to-peer collaborations in the web services area. One of the important problems related to WS-CDL is the lack of separation between its meta-model and its syntax. A formal semantics can provide validation capabilities for WS-CDL.

In this paper, we define a simple language CDL which covers the features of WS-CDL related to the participant roles and the collaborations among roles. A formal operational semantics for the language is presented. Based on the semantics, we can apply model-checking technique to verify the correctness of specified systems. Given a system, we might check its consistency, and various properties (e.g. no deadlock), and the satisfaction with business constraints. We also give an example of a purchase order choreography to show how to verify properties based on our model.

Towards the semantics and verification of full WS-CDL, CDL focuses on just a few key issues related to web service choreography. The goal in the designing of CDL is to make the proof of its properties as concise as possible, while still capturing the core features of WS-CDL. The features of WS-CDL that CDL does model include roles, variables, activities(control-flow, workunit, skip, assignment, interaction) and choreography. CDL omits some advanced features such as some details of the channel, exception and finalize blocks. Other features missing from CDL include base types(relationship type, participant type, information type), token, token locator, expressions and some basic activities such as silent and perform. Extending CDL to include more features of WS-CDL will be one direction of our further work.

For future work, we want to integrate the exception handling and finalize block mechanisms into our model, which are important facilities to support long-running interaction in WS-CDL. This can help us to capture and understand the full control flow of the WS-CDL specification, and to form a more complete semantics of the language. We are also trying to find more interesting properties that can be verified under our framework. Moreover, based on our previous work on the semantics of BPEL [7,18], we are considering to compare these two semantic models, and related future work includes: (1) to project a global CDL model into several BPEL models on different roles; (2) to check that whether a given BPEL model conforms to the global CDL model.

Acknowledgements

We would like to thank Dai Xiwu for many helpful comments.

References

1. World wide web consortium. http://www.w3.org/.
2. Business process execution language for web services, version 1.1. May 2003. http://www-106.ibm.com/developerworks/webservices/library/ws-bpel.
3. A.Brogi, C.Canal, E.Pimentel, and A.Vallecillo. Formalizing web service choreography. In *WS-FM 2004*. Electronic Notes in Theoretical Computer Science, 2004.
4. Assf Arkin. Business process modeling language. November 2002. http://www.bpmi.org/.
5. Alistair Barros, Marion Dumas, and Phillipa Oaks. A Critical Overview of the Web Services Choreography Description Language. 2005. http://www.bptrends.com.
6. F.Leymann. Web services flow language (wsfl 1.0). Technical report, IBM, May 2001.
7. Pu Geguang, Zhao Xiangpeng, Wang Shuling, and Qiu Zongyan. Towards the semantics and verification of bpel4ws. In *International Workshop on Web Languages and Formal Methods, WLFM2005*. to appear in Electronic Notes in Theoretical Computer Science, Elsevier 2006.
8. H.Foster, S.Uchitel, J.Magee, and J.Kramer. Model-based verification of web service compositions. In *18th IEEE International Conference on Automated Software Engineering (ASE'03)*. IEEE Computer Science, 2003.
9. Gerard J. Holzmann. *The SPIN Model Checker:Primer and Reference Manual*. Addison-Wesley, 2003.

10. Jayadev Misra. A programming model for the orchestration of web services. In *Software Engineering and Formal Methods(SEFM'04)*. IEEE Computer Science, 2004.
11. N.Busi, R.Gorrieri, C.Guidi, R.Lucchi, and G.Zavattaro. Towards a formal framework for choreography. 2005.
12. Nickolaos.Kavantzas. Aggregating web services: Choreography and ws-cdl. Technical report, Oracle Coporation, 2004.
13. N.Kavantzas, D.Burdett, G.Ritzinger, T.Fletcher, Y.Lafon, and C.Barreto. Web Services Choreography Description Language Version 1.0. November 9,2005. http://www.w3.org/TR/2005/CR-ws-cdl-10-20051109/.
14. Guus Ramackers. Web services choreography description language(ws-cdl). Technical report, Oracle Coporation, 2004.
15. Steve Ross-Talbot. Web services choreography and process algebra. 29th April 2004.
16. Gwen Salaun, Lucas Bordeaux, and Marco Schaerf. Describing and reasoning on web services using process algebra. In *2nd International Conference on Web Services*. IEEE, 2004.
17. S.Thatte. Xlang: Web services for business process design. Technical report, Microsoft, 2001.
18. Qiu Zongyan, Wang Shuling, Pu Geguang, and Zhao Xiangpeng. Semantics of bpel4ws-like fault and compensation handling. In *FM2005,LNCS 3582*. Springer, 2005.

Appendix: SPIN Code

```
/* A case study of purchase order service
 * po = purchaseOrder, cc = creditCheck, ic = inventoryCheck
 *
 * Spin doesn't support named enum type.
 * All of the types are "mtype", but the comment at the end of
 * each line specifies the name.
 */

/* types declaration */
mtype = { po, poAck, poReject, poResponse, ccReq, ccResp, icReq, icResp};
/* link variables, here implemented as enum constants */
mtype = { sent, received, complete }; /* state */
mtype = { Ok, notOk }; /* resp */
mtype = { finished_doIC, finished_doCC }; /* para aux */

/* channels declaration */
chan buyer_to_seller = [0] of { mtype };
chan seller_to_buyer = [0] of { mtype };
chan seller_to_credit = [0] of { mtype };
chan seller_to_inventory = [0] of { mtype };
chan credit_to_seller = [0] of { mtype, mtype };
chan inventory_to_seller = [0] of { mtype, mtype };
chan para_aux = [0] of { mtype };

/* variables declaration */
mtype buyer_poState;
mtype seller_ccResp, seller_icResp;
mtype seller_poState, seller_ccState, seller_icState;
mtype credit_ccState;
mtype inventory_icState;

mtype msg;
/* auxiliary variable, recording the last sent message
 * through channel seller_to_buyer
 */

/* roles declaration */
active proctype buyer() {
  buyer_to_seller ! po;
```

```
  buyer_poState = sent;
  seller_to_buyer ? poAck;
  if
    :: seller_to_buyer ? poRej -> buyer_poState = complete
    :: seller_to_buyer ? poResp -> buyer_poState = complete
  fi
}

active proctype seller() {
  buyer_to_seller ? po;
  seller_poState = received;
  seller_to_buyer ! poAck;
  run doCC(); run doIC(); /*inventory check & credit check */
  if
    :: para_aux ? finished_doIC -> para_aux ? finished_doCC
    :: para_aux ? finished_doCC -> para_aux ? finished_doIC
  fi; /* wait until both processes are finished */

  if
    :: (seller_icResp == Ok && seller_ccResp == Ok) ->
          atomic {
            seller_to_buyer ! poResp;
            msg = poResp;
          }
          seller_poState = complete;
    :: (seller_icResp == notOk || seller_ccResp == notOk) ->
          atomic {
            seller_to_buyer ! poRej;
            msg = poRej;
          }
          seller_poState = complete;
  fi
}

proctype doCC() {
  seller_to_credit ! ccReq;
  credit_to_seller ? ccResp(seller_ccResp);
  seller_ccState = received;
  /* indicate process finished */
  para_aux ! finished_doCC;
}

proctype doIC() {
  seller_to_inventory ! icReq;
  inventory_to_seller ? icResp(seller_icResp);
  seller_icState = received;
  /* indicate process finished */
  para_aux ! finished_doIC;
}

active proctype credit() {
  seller_to_credit ? ccReq;
  if
    :: credit_to_seller ! ccResp(Ok)
    :: credit_to_seller ! ccResp(notOk)
  fi;
  credit_ccState = sent;
}

active proctype inventory() {
  seller_to_inventory ? icReq;
  if
    :: inventory_to_seller ! icResp(Ok);
    :: inventory_to_seller ! icResp(notOk);
  fi;
  inventory_icState = sent;
}
```