Towards the Formal Model and Verification of Web Service Choreography Description Language

Towards the Formal Model and Verification of Web Service Choreography Description Language [⋆]

Xiangpeng Zhao, Hongli Yang, and Zongyan Qiu

LMAM and Department of Informatics, School of Math.,
Peking University, Beijing 100871, China
{zxp,yhl,qzy}@math.pku.edu.cn

Abstract. The Web Services Choreography Description Language (WS-CDL) is a W3C specification for the description of peer-to-peer collaborations of participants from a global viewpoint. For the rigorous development 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 the simplified WS-CDL, which includes important concepts related to participant roles and collaborations among them in a choreography. The formal operational semantics of CDL is given. Based on the formal model, we discuss further: 1) project a given choreography to orchestration views, which provides a basis for the implementation of the choreography by code generation; 2) translate WS-CDL to the input language of the model-checker SPIN, which allows us to automatically verify the correctness of a given choreography. An automatic translator has been implemented.

1 Introduction

Web services 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 [2] candidate recommendation for web service composition. WS-CDL is an XML-based language for the describing peer-to-peer collaborations of participants by defining, based on a global viewpoint, from their common and complementary observable behavior [18]. WS-CDL is neither an "executable business process description language" nor an implementation language. The execution logic of the application is covered by languages at another level, such as XLANG [21], WSFL [7], BPEL [3], BPML [5], 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.

⋆ Supported by National Natural Science Foundation of China (No. 60573081)

As discussed in [6], WS-CDL lacks the separation between its meta-model and its syntax, and lacks of 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 collaborated with each other. To guarantee the correct interaction of independent, communicating web services becomes even more critical in the open-end world of web services [20]. 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 provide a grounding for 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 1) to generate orchestration views from a given choreography; 2) to reason about the properties that should be satisfied by the specified system. We propose a projection function for orchestration generation and discussed the correctness issue of the projection. We also provide an automatic translation tool which can convert a choreography into the input language of the model checker SPIN [12]. Afterwards, we can either simulate or verify the choreography automatically. Besides, manual reasoning based on our model is also possible, as discussed in our previous report [13].

This paper is organized as follows: Section 2 is an informal overview of WS-CDL. We present the formal model of CDL in Section 3, including its syntax and operational semantics. In Section 4 we discuss the projection from choreography to orchestration, while in Section 5 we discuss how to model-check WS-CDL specification using our automatic translator. Some related work is discussed in Section 6, and section 7 concludes.

2 Overview of WS-CDL

This section provides an overview of WS-CDL, as defined in WS-CDL specification [18] 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 two types of activities in WS-CDL: basic activities and control-flow activities.

Basic activities include a *noAction* action, which does not do anything; an *assign* activity, which assigns, within one role, the value of one variable or an expression to another variable; and 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 interaction activity is composed of: 1) the participant roles involved; 2) the exchanged information and the corresponding direction(s); 3) the observable information changes; 4) 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>
```

The control-flow activities include sequence, parallel, choice and workunit. The *sequence*, *parallel* and *choice* activities have similar meanings as in the other programming languages. A *workunit* describes the conditional and repeated execution of an activity [6].

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 are no independent variables in WS-CDL, i.e. each variable must belong to some role.

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:

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

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 $\overline{x}$. Note that e must only contain variables that belong to the same role r. For remote assignments (i.e. assign some variable of one role to some variable of another role), we must use the interaction activity, which 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. After the operation, some state change will be performed by rec, which is the shorthand for the assignments $f.\overline{x} := \overline{e_1}, t.\overline{y} := \overline{e_2}$. Here $\overline{x}$ and $\overline{y}$ are two lists of state variables on the roles f and t respectively.

The syntax of the control-flow activities is listed here:

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

An activity is either a basic activity BA, a workunit, or a control-flow activity. The workunit introduced in WS-CDL is 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 $g_1 \Rightarrow A \mathbin{[\!]} g_2 \Rightarrow B$, or a parallel activity $A \parallel B$.

The WS-CDL specification includes many well-formedness and typing rules, such as "In an interaction, each information exchange variable has the same type on the sender and the receiver". We are developing a type system for CDL to statically verify the validity of all these rules. In this paper we do not consider well-formnedness problems, and assume that the CDL program under consideration is always 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

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

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. Note that every variable appearing in $\overline{e}$ should belong to r, i.e. remote value fetch is not allowed.

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

In an interaction activity, some information may exchange between two participant roles, namely a "from" role f and a "to" role t. After the operation op is accomplished, there may be some variable updates on both roles according to the assignments in rec. The semantics for op is not defined here; we can view op as an external atomic activity. As a result, we can define the trace of a choreography as the sequence of operations it performs.

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

$$\langle \mathsf{comm}\ (f.x \overset{c}{\leftarrow} t.y, rec, op),\ \sigma\rangle \longrightarrow \langle op; rec, \sigma[f.x := t.y]\rangle \qquad \text{(RESP)}$$

$$\langle \mathsf{comm}\ (f.x \overset{ct}{\rightarrow} t.y, f.u \overset{cf}{\leftarrow} t.v, rec, op),\ \sigma\rangle \longrightarrow \langle op; rec, \sigma[t.y := f.x, f.u := t.v]\rangle \qquad \text{(REQ-RESP)}$$

Control-flow Activity 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.

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

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

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.

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

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

The workunit activity $(g : A : p)$ is blocked when the guard condition g evaluates to false. When g evaluates to true, A is executed. After the execution, repetition condition p is tested. If p evaluates to false, then the activity terminates; if true, then the workunit restarts. In the WS-CDL syntax, g and p can be omitted. An omitted condition means that it is always true.

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

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 $A; B$.

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

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

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.

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

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

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.

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

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

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

We use interleaving semantics for the parallel composition:

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

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

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

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

Based on the formal semantics, we can do manual reasoning about the properties that should be satisfied by a given choreography. A purchase order choreography example is given in our previous work [13]. In this paper, we focus on automatic verification using model-checking, as described in Section 5.

4 Projection from CDL to Orchestration Views

As we have described, CDL provides a choreographical view of the interacting web services, which involves the interaction of many parties. For code generation, simulation and verification purposes, it is meaningful to generate orchestration views from a given choreography view, while each of them describes only the interaction behavior of one party with its related partners. In our example, the credit checker only deals with the seller, not the buyer nor the inventory. An orchestration view for the credit checker is a projection from the choreography that hides the behavior of all the unrelated parties.

In choreography, as at a higher level of view, we can always let the two-party interaction start without waiting, as explained informally by the following:

$$\mathsf{comm}\,(f.x \overset{ch}{\rightarrow} t.y, rec, op) \;\approx\; (ch!f.x \parallel ch?t.y) \;\approx\; t.y := f.x$$

In other words, the data transportation through channels is implicit under a choreography view. This explains the reason that we use variable assignment for the semantics of interaction. At the local view level, we remove the interaction activities from our syntax, while adding the channel communication activities:

$$BA_{orc} ::= \mathsf{skip} \mid f.\overline{x} \;:=\; t.\overline{y} \mid ch!f.x \mid ch?t.y$$

The semantics of an orchestration is similar to a process in an ordinary process algebra, and we do not want to discuss the details here. Note that according to WS-CDL specification, we can only send some variable through a channel, rather than an arbitrary expression.

We define a set of projection rules from a given choreography to a process of a given target role r_t as a function Pr, which maps an activity in choreography to an activity in the local view. For a choreography $C[\overline{r}\ ,\ A]$ that has a root activity A and n roles $r_1, \cdots, r_n$, we define the projection from C to role r_i as $Pr(A, r_i)$.

$$Pr(\mathsf{skip}, r_t) \overset{def}{=} \mathsf{skip}$$

$$Pr(r.\overline{x}\ :=\ \overline{e}, r_t) \overset{def}{=} \begin{cases} r.\overline{x}\ :=\ \overline{e} \text{ if } r_t = r \\ \mathsf{skip} \qquad \text{otherwise} \end{cases}$$

$$Pr(\mathsf{comm}\ (f.x \xrightarrow{c} t.y, rec, op), r_t) \overset{def}{=} \begin{cases} c!f.x; Pr(rec, r_t) & \text{if } r_t = f \\ c?t.y; op; Pr(rec, r_t) & \text{if } r_t = t \\ \mathsf{skip} & \text{otherwise} \end{cases}$$

$$Pr(\mathsf{comm}\ (f.x \xleftarrow{c} t.y, rec, op), r_t) \overset{def}{=} \begin{cases} c?t.y; op; Pr(rec, r_t) & \text{if } r_t = f \\ c!f.x; Pr(rec, r_t) & \text{if } r_t = t \\ \mathsf{skip} & \text{otherwise} \end{cases}$$

$$Pr(\mathsf{comm}\ (f.x \xrightarrow{c_t} t.y, f.u \xleftarrow{c_f} t.v, rec, op), r_t) \overset{def}{=} \begin{cases} c_t!f.x; c_f?t.y; Pr(rec, r_t) & \text{if } r_t = f \\ c_t?f.x; op; c_f!t.y; Pr(rec, r_t) & \text{if } r_t = t \\ \mathsf{skip} & \text{otherwise} \end{cases}$$

$$Pr(A; B, r_t) \overset{def}{=} Pr(A, r_t); Pr(B, r_t)$$

$$Pr(p?A, r_t) \overset{def}{=} p?Pr(A, r_t)$$

$$Pr(p * A, r_t) \overset{def}{=} p * Pr(A, r_t)$$

$$Pr(g\!:\!A\!:\!p, r_t) \overset{def}{=} g\!:\!Pr(A, r_t)\!:\!p$$

$$Pr(A \sqcap B, r_t) \overset{def}{=} Pr(A, r_t) \sqcap Pr(B, r_t)$$

$$Pr(g_1 \Rightarrow A \,[\!]\, g_2 \Rightarrow B, r_t) \overset{def}{=} g_1 \Rightarrow Pr(A, r_t) \,[\!]\, g_2 \Rightarrow Pr(B, r_t)$$

$$Pr(A \parallel B, r_t) \overset{def}{=} Pr(A, r_t) \parallel Pr(B, r_t)$$

Only basic activities are modified by the projection, because the projection should not change the control flow. Intuitively, if some activity is not related to the given role and should be hidden, we simply replace it with skip. For interaction activities, we will replace them with channel communications according to the direction of the interaction. The control-flow activities that do not have guards are not modified.

Given a choreography, we can project it to each involved role $r \in \overline{R}$ with the rules above. After that, we can consider implementation, simulation or verification. Note that the generated orchestration may contain a lot of redundant skip, and some redundant structured activities. We have removed most of the redundancy in our automatic translator which is described in the next section.

For any choreography $C[\overline{r}\ ,\ A]$ that has n roles $r_1, \cdots, r_n$, the projection function Pr satisfies the following equation:

$$(Pr(A, r_1)\ \parallel\ Pr(A, r_2)\ \parallel \cdots \parallel\ Pr(A, r_n)) =_{sem} A \qquad (1)$$

It means that if we combine the projected behaviors together, then the resulted system should have the same behavior as the choreography, i.e. they are equal according to the semantics definition. The formal proof of the equation is based on structural induction on the definition of Pr. The full proof is omitted here due to the space limitation.

Generation to BPEL code is an important concern in the design of WS-CDL. Following the the projection rules presented here, it is easy to implement a translator from WS-CDL to the code for each role as a BPEL interface, whose correctness is assured by the Equation 1. We list the implementation of such a translator as one of our future work.

5 Model-checking CDL Specification

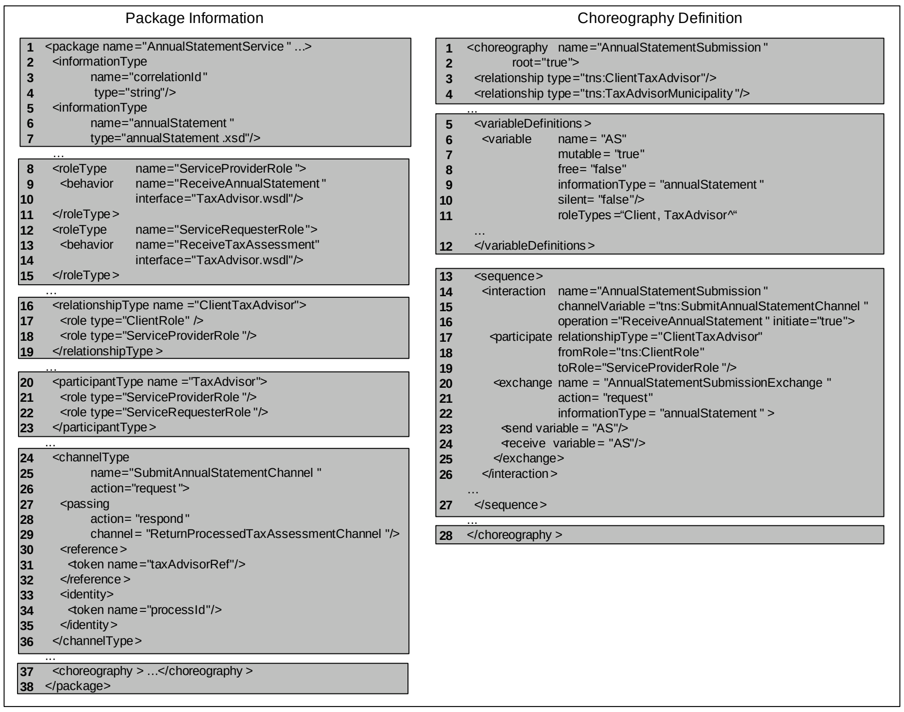
In this section we discuss how to verify a given WS-CDL specification using the SPIN model checker [12]. The input language of SPIN is called Promela, which is a language for modeling finite-state concurrent processes. SPIN can verify or falsify (by generating counterexamples) LTL (Linear Temporal Logic) properties of Promela specifications using an exhaustive state space search. Given the XML specification of a CDL choreography, we generate a Promela specification, which consists of some communicating concurrent processes denoting different parties. We implement the translation in two steps: (1) from WS-CDL specification to projected orchestration specifications; (2) from the orchestration specifications to Promela processes with communication channels. The first phase has been discussed in the previous section, and we will focus on the second phase in this section.

5.1 Translation to Promela

We discuss our translation procedure using the annual tax statement example taken from [14]. In the choreography, the client asks an advisor to help him pay tax to the municipality according to the annual statement he provides. The client's request may be rejected directly by the advisor, or forwarded to the municipality, which will return a notification that either accepts the statement or rejects it. The WS-CDL source code is 300-lines-long and is difficult to read. An abridged version of the source code is shown in Figure 1.

We give the translated Promela code of this example in Figure 2. The first part of the code consists of some type declarations and variable declarations. We introduce a variable named r_x for each variable x under role r. Each information type variable is converted into a `mtype` variable; while each channel variable is converted into a channel `chan`. Although model-checkers have proved to be powerful in verifying the control flow of a system, it performance is quite poor when we allow the variables to have a wide range of possible values. As a prototype system, we only consider two states `nil` and `something` of each information variable, which denotes whether the variable has been assigned by some value or not. We also introduce some auxiliary boolean variables to implement parallelism, which are discussed later.

The second part of the code consists of several processes that denotes roles in the choreography. The `init` process instantiates three processes corresponding to each


Package Information
Choreography Definition

```
<package name="AnnualStatementService " ...>
  <informationType
            name="correlationId"
             type="string"/>
  <informationType
            name="annualStatement "
            type="annualStatement .xsd"/>
   ...
  <roleType       name="ServiceProviderRole ">
    <behavior     name="ReceiveAnnualStatement "
                  interface="TaxAdvisor.wsdl"/>
  </roleType >
  <roleType       name="ServiceRequesterRole ">
    <behavior     name="ReceiveTaxAssessment"
                  interface="TaxAdvisor.wsdl"/>
  </roleType >
  ...
  <relationshipType name ="ClientTaxAdvisor">
    <role type="ClientRole" />
    <role type="ServiceProviderRole "/>
  </relationshipType >
  ...
  <participantType name ="TaxAdvisor">
    <role type="ServiceProviderRole "/>
    <role type="ServiceRequesterRole "/>
  </participantType >
  ...
  <channelType
            name="SubmitAnnualStatementChannel "
            action="request ">
    <passing
            action= "respond"
            channel= "ReturnProcessedTaxAssessmentChannel "/>
    <reference >
      <token name="taxAdvisorRef"/>
    </reference >
    <identity>
      <token name="processId"/>
    </identity >
  </channelType >
  ...
  <choreography > ...</choreography >
</package>
```

```
<choreography   name="AnnualStatementSubmission "
          root="true">
  <relationship type="tns:ClientTaxAdvisor"/>
  <relationship type="tns:TaxAdvisorMunicipality "/>
 ...
  <variableDefinitions >
    <variable     name= "AS"
                  mutable= "true"
                  free= "false"
                  informationType = "annualStatement "
                  silent= "false"/>
                  roleTypes ="Client , TaxAdvisor^"
   ...
  </variableDefinitions >

  <sequence >
    <interaction  name="AnnualStatementSubmission "
                  channelVariable ="tns:SubmitAnnualStatementChannel "
                  operation ="ReceiveAnnualStatement " initiate="true">
      <participate relationshipType ="ClientTaxAdvisor"
                  fromRole="tns:ClientRole"
                  toRole="ServiceProviderRole "/>
      <exchange   name = "AnnualStatementSubmissionExchange "
                  action= "request"
                  informationType = "annualStatement " >
        <send variable = "AS"/>
        <receive  variable= "AS"/>
      </exchange>
    </interaction >
  ...
  </sequence >
 ...
</choreography >
```

Fig. 1. WS-CDL Specification Example

role. Based on the projection technique introduced in Section 4 we can give the execution logic of each process. Assuming the projection has been done, in Table 1 we give a mapping from orchestration specification of role r to Promela code. Since Promela supports most of the activities defined in our CDL semantics, most translation is quite straightforward.

In Promela, `if` statement is a blocking guarded choice. The system can proceed only if at least one guard is satisfied. If more than one guards are satisfied, then the system will make a non-deterministic choice. The `do` statement is used for repeating, and is similar with `if`. For parallel activities, we first introduce some auxiliary processes with the prefix "para" for each block in the parallel activity, and then call the processes to start by a `run` statement. Since `run` is an asynchronous call in Promela, we need some extra mechanism to make the calling process wait until all the called processes have finished running. The auxiliary variables with prefix `r_para_aux` are introduced for this purpose. We use conditional expressions such as `r_para_aux_A == true` to block the execution of the calling process. The auxiliary variables such as `r_para_aux_A` are assigned by `true` only at the end of each called process, thus achieving the synchronous calling mechanism.

Finally, we add a label such as `TaxAdvisor_end:` at the end of each process, which is useful for expressing properties during verification.

```
/* Choreography: AnnualStatementSubmission */
mtype = {nil, something, };
chan SubmitAnnualStatementChannel = [0] of { mtype };
chan ReturnNotificationChannel ...
mtype ServiceProviderRole_AS = nil;
mtype ClientRole_AS = nil; ...
mtype ServiceProviderRole_RN = nil; ...
bool Client_para_aux_1 = false;
...
proctype TaxAdvisor() {
  SubmitAnnualStatementChannel ? ServiceProviderRole_AS;
  if
  :: ReturnNotificationChannel ! ServiceProviderRole_RN;
  :: {run TaxAdvisor_para_1();      /* parallel begin */
      run TaxAdvisor_para_2();
      TaxAdvisor_para_aux_1 == true;
      TaxAdvisor_para_aux_2 == true; /* parallel end */
      ReturnTaxAssessmentChannel ? ServiceRequesterRole_TA;
      ReturnProcessedTaxAssessmentChannel ! ServiceProviderRole_PTA; }
  fi;
  ServiceProviderRole_AS = something;
  TaxAdvisor_end: skip; /* ending label */
}
proctype Client() {...}
proctype Municipality() {...}
/* parallel auxiliary proctypes*/
proctype TaxAdvisor_para_1() {
  ReturnNotificationChannel ! ServiceProviderRole_AC;
  TaxAdvisor_para_aux_1 = true;
}
proctype Client_para_1() ...
init { atomic {
    run Client();
    run TaxAdvisor();
    run Municipality();
}}
```

Fig. 2. Promela Code Snippet

5.2 Simulation and Verification

We have implemented an automatic translator[1] according to the translation rules using the XSLT (eXtensible Stylesheet Language Transformation) language. XSLT is a language designed for translating a source XML data into some target data. It is a functional programming language which supports pattern-matching, and has a built-in parser for XML data. With these features and the formal defined semantics, it is not very hard to implement the translator. In the implementation, we reuse some source codes provided in [14]. We have also implemented a Java based user interface as the front-end of the translator. The user can easily translate a WS-CDL specification into a Promela file, and then simulate or verify the Promela processes with SPIN.

With our translator, the user can view the choreography in a graphical and interactive way in the simulator provided in SPIN. A simple simulation scenario of the buyer-seller example given in the WS-CDL specification [18] is shown in Figure 3.

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

[1] The tool can be downloaded at http://www.is.pku.edu.cn/∼fmows/.

skip	`skip`
$r.x := e$	`r_x = e`
$ch!e$	`ch ! e`
$ch?r.x$	`ch ? r_x`

$p?A$	`if`
	`:: p -> A`
	`:: !p-> skip`
	`fi`

$p * A$	`do`
	`:: p -> A`
	`:: !p-> break`
	`od`

$g\!:\!A\!:\!p$	`do`
	`:: g -> A; if :: p->skip :: !p->break fi`
	`:: !g-> break`
	`od`

$A; B$	`A; B`

$g_1 \Rightarrow A \,[\!]\, g_2 \Rightarrow B$	`if`
	`:: g1 -> A`
	`:: g2 -> B`
	`fi`

$A \sqcap B$	`if`
	`:: A`
	`:: B`
	`fi`

$A \parallel B$	`atomic {`
	`run r_paraA();`
	`run r_paraB();`
	`};`
	`r_para_aux_A == true;`
	`r_para_aux_B == true;`

Table 1. Translation to Promela for Role r

– The system will never deadlock. (`timeout` is a reserved word in Promela)

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

– Every role will always reach the ending state eventually.

```
[] (Client@Client_End && TaxAdvisor@TaxAdvisor_End &&
    Municipality@Municipality_End)
```

– The client can always finish running while receiving the processed tax assessment (PTA) from the municipality. This property is not satisfied by the choreography, since the advisor may refuse the request. SPIN can detect the false property, and generate a counter example to illustrate the reason for the failure in a graphical way.

```
[] (Client@Client_End && ClientRole_PTA != nil)
```

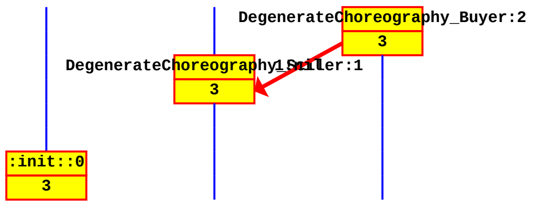

DegenerateChoreography_Buyer:2
3
DegenerateChoreography1Seller:1
3
:init::0
3

Fig. 3. A Simple Simulation Scenario

– If the processed annual statement (PAS) is forwarded to the municipality, then the client will always receive the processed tax assessment eventually. Such acknowledgement is very useful in designing protocols.

```
[] (ServiceRequesterRole_PAS != nil -> (<> ClientRole_PTA != nil)
```

We have translated and verified several examples, including the buyer-seller example given in the WS-CDL specification [18], the purchase order process given in [13], and the annual tax statement process proposed in [14]. The verification procedure is speedy; for each example, it only costs several seconds on a Pentium 4 machine with 512MB memory.

6 Related Work and Discussion

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 *et al.* [10,11] discussed a model-based approach to verify web service compositions and developed the tool LTSA that translates BPEL or WS-CDL specification to the FSP process algebra model. G.Salaun *et al.* developed a process algebra to derive the interactive behavior of a business process out from a BPEL specification [20]. 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 [4]. X.Fu *et al.* gave a translation from BPEL to Promela [8]. Pi4SOA [1] is a tool for designing WS-CDL choreography with a nice graphical user interface, and supports projection from WS-CDL to BPEL or Java. It has a text-based simulator which is relatively difficult to use, and does not provide verification mechanism. There are also works on the formal semantics of web services languages. In our previous work, we presented an operational semantics to a simplified version of BPEL with some important concepts related to fault and compensation handling [9,22].

In a recent paper [16], N.Busi *et al.* proposed a simple choreography language whose main concepts are based on WS-CDL. Different from our language, this language splitted the request and response message exchange in one interaction, and there were no state record variables. Also, it did not consider the verification of web service properties. Misra [15] proposed a new programming model for the orchestration of web services. It is relatively far from practice and needs further investigation. Mendling and Hafner [14] proposed a translation algorithm from BPEL to WS-CDL. Since they did not provide a formal model, the correctness of the translation remained to be proved.

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 [17,19].

We have noticed some debatable issues about the guard expressions during our investigation of WS-CDL. According to the specification, if we write a workunit $g : A : p$ with the guard condition $g = (r_1.x = 1 \wedge r_2.y = 2)$ that involves more than one role (it is called a "globalized view" in the specification), then the execution of A on r_1 only requires that $r_1.x = 1$, because r_1 cannot directly fetch the state of y on r_2. Similarly, the execution of r_2 only depends on the sub-formula related to r_2 in g.

This guard partition semantics is somehow confusing from our intuitive idea of a guard condition. It can also bring some difficulty for the choreography designer: a workunit with the form of $r_1.x = 1 : \mathsf{comm}\,(r_1.x \xrightarrow{c} r_2.y, rec, op) : true$ may be troublesome, since r_1 may send a few messages and then terminate according to the guard $r_1.x = 1$, while r_2 will wait for infinite number of messages because it does not have a guard. Therefore, a deadlock is inevitable. Moreover, even if we define the guard as $r_1.x = 1 \wedge r_2.y = 2$, we are still not sure whether the choreography is deadlock-free, because $r_1.x = 1$ and $r_2.y = 2$ may not become true(or false) simultaneously. Similarly, although the conditional activity $(r_1.x = 1)?r_2.y := 1$ is permitted by the specification, it is usually not a desirable choreography.

In [1,14], after automatic projection to BPEL, blank guards for those unmentioned roles are generated for the user to fill in, which is obviously not a simple task. It is not very hard to partition the guard conditions for each roles and integrate the resulted sub-conditions into the framework discussed in this paper. But, as we think, the mechanisms proposed in the current version of WS-CDL is not very satisfiable. Some clearer and more understandable structures should be investigated and designed in the future, which is a very interesting future work. In this case, we have not included the guard partitioning problem in this paper.

7 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 discussed how to 1) project a given choreography to orchestration views, which provides a basis for the implementation of the choreography by code generation; 2) apply model-checking technique to automatically verify the correctness of a given choreography. Given a system, we might check its consistency, and various properties (e.g. no dead-lock), and the satisfaction with business constraints. We have developed an automatic translation tool from WS-CDL to the input language of the model checker SPIN. The user only needs to provide a WS-CDL specification, and then do simulation or verification automatically. We have tried several cases using our translation tool, and managed to verify some useful properties.

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. The conformance problem between orchestration and choreography, i.e. whether some given BPEL orchestration processes are consistent with the given WS-CDL choreography model, also needs further investigation. The Equality 1 proposed in Section 4 can be viewed as a first step towards the formal definition of the conformance concept.

Acknowledgements

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

References

1. Pi4soa. http://www.pi4soa.org/.
2. World wide web consortium. http://www.w3.org/.
3. Business process execution language for web services, version 1.1. May 2003. http://www-106.ibm.com/developerworks/webservices/library/ws-bpel.
4. A.Brogi, C.Canal, E.Pimentel, and A.Vallecillo. Formalizing web service choreography. In *WS-FM 2004*. Electronic Notes in Theoretical Computer Science, 2004.
5. Assf Arkin. Business process modeling language. November 2002. http://www.bpmi.org/.
6. Alistair Barros, Marion Dumas, and Phillipa Oaks. A Critical Overview of the Web Services Choreography Description Language. 2005. http://www.bptrends.com.
7. F.Leymann. Web services flow language (wsfl 1.0). Technical report, IBM, May 2001.
8. X. Fu, T. Bultan, and J. Su. Analysis of interacting bpel web services. In *Proc. of WWW'04, pp. 621-630*, 2004.
9. 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.

10. 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.
11. H.Foster, S.Uchitel, J.Magee, and J.Kramer. Model-based analysis of obligations in web service choreography. In *Proc. of International Conference on Internet and Web Applications and Services 2006*, 2006.
12. Gerard J. Holzmann. *The SPIN Model Checker:Primer and Reference Manual*. Addison-Wesley, 2003.
13. Yang Hongli, Zhao Xiangpeng, and Qiu Zongyan. A formal model of web service choreography description language(ws-cdl). Technical report, Department of Informatics, School of Math., Peking University, China., January 2006. available at: http://www.math.pku.edu.cn/yanghl/papers/CDL.pdf.
14. J. Mendling and M. Hafner. From inter-organizational workflows to process execution: Generating bpel from ws-cdl. 2005.
15. Jayadev Misra. A programming model for the orchestration of web services. In *Software Engineering and Formal Methods(SEFM'04)*. IEEE Computer Science, 2004.
16. N.Busi, R.Gorrieri, C.Guidi, R.Lucchi, and G.Zavattaro. Towards a formal framework for choreography. 2005.
17. Nickolaos.Kavantzas. Aggregating web services: Choreography and ws-cdl. Technical report, Oracle Coporation, 2004.
18. 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/.
19. Steve Ross-Talbot. Web services choreography and process algebra. 29th April 2004.
20. 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.
21. S.Thatte. Xlang: Web services for business process design. Technical report, Microsoft, 2001.
22. Qiu Zongyan, Wang Shuling, Pu Geguang, and Zhao Xiangpeng. Semantics of bpel4ws-like fault and compensation handling. In *FM2005,LNCS 3582*. Springer, 2005.