Model-checking of Web Services Choreography*

Yang Hongli\(^1\), Zhao Xiangpeng\(^2\), Cai Chao\(^2\), and Qiu Zongyan\(^2\)

\(^1\) College of Computer Sciences
Beijing University of Technology, Beijing 100022, China
\(\text{yhl@bjut.edu.cn}\)

\(^2\) LMAM and Department of Informatics, School of Math.,
Peking University, Beijing 100871, China
\{zxp,caic,qzy\}@math.pku.edu.cn

Abstract Web services choreography describes the global model of service interactions among a set of participants. In order to achieve a common business goal, the protocols of interaction must be correct. In this paper, we model interactions with recordings of state/channel variable changes that can occur as a result of carrying out the interactions. Thus, it is possible to verify not only normal control flow properties such as deadlock-freeness, but also channel-passing related problems such as channel-absence. Concretely, we propose a small language CDL, together with an operational semantics. We illustrate with examples how service choreographies can be specified in CDL, and how the verification can be carried out using the SPIN model-checker.

Keywords: Choreography, Formal Model, Model-checking

1 Introduction

Web services promise the inter-operability of various applications running on heterogeneous platforms over the Internet. Web services composition refers to the process of combining web services to provide value-added services, 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 description of the single services, possibly with cooperation of other services, is called an orchestration. The de facto standard for orchestration is BPEL [2] (Web Services Business Process Execution Language) developed by OASIS, a consortium comprising BEA, IBM, Microsoft etc. The global view of the interactions are described by the so-called choreography. WS-CDL (Web Services Choreography Description Language) [1] is a W3C candidate recommendation, designed for describing the common and collaborative observable behavior of multiple services that interact with each other to achieve a common (business) goal.

In large service-oriented systems, stockholders may require a global picture of the way by which the related services interact with each other, rather than multiple local

\* Supported by open foundation of State Key Laboratory of Computer Science, Institute of Software, Chinese Academy of Sciences (No. SYSKF0703), and National Natural Science Foundation of China (No. 60603033 and No. 60773161)
pictures of individual services [25]. Since choreography describes the interaction protocol among multiple participants in a global-view manner, it can be used to guide the development of the participants. Choreography has attracted much interest in the research field, including the model, analysis and implementation of choreography.

For the participants to carry out the collaboration, they must communicate with one another. In an ideal situation, different participants of a business process know one another, that is, they know all the channels used in their communication during the work, before the work starts. When all channels are known by their users and all the usage of the channels are described statically in the business process specification, we say that the process has a static communication structure. Most formal work on service composition adopted this assumption. However, in real world applications, the static communication structure may not be sufficient. In many practical situations, some participants of a multi-party business process may be selected dynamically during the execution by some participant already in the work. Also, a participant may not take part in a process until some specific event happens in the execution. If a participant needs to join the interaction dynamically during the execution and then communicate with the others, it must obtain channels from the participants which are already in the work. Thus, in general, issues relevant to channel passing are indispensable in choreography.

In order to guarantee the correct interaction among a set of independent, communicating services, it is important to verify that the choreography specification is designed correctly. In recent years, there are some literatures about the modeling of choreography [8,14,9]. Based on our knowledge, most work only focus on the control flow, and few work focus on verifying data-related properties in choreography level, especially channel-related properties. In order to model-check such properties, the key is to choose a suitable abstract model of choreography. If the choreography model is over-simplified, it may too abstract to provide useful results. On the other side, the verification problem may be undecidable in general [20] due to infinite state space caused by variables.

To model choreography in a suitable way, we focus on the fact that the interactions can be described from the viewpoint of an ideal observer who oversees all interactions among a set of services [25]. In an interaction, state variable and channel variable changes can be recorded [1]. The state variables are useful for determining the decisions and actions to be taken within a choreography, while channel variables contain information such as the URL to which the message could be sent. Particularly, the channel variable changes reflect some channel passings happened during executing an interaction. Since state and channel variables only take finite values, we can check if a complex business protocol expressed as a choreography satisfies some channel-related properties.

Based on these recognitions, we propose a small Choreography Description Language CDL as a formal framework in this paper. CDL is inspired by WS-CDL, which allows recordings of state and channel variable changes in interactions, as well as in normal assignments. The formal syntax and operational semantics of CDL are defined here, and some interesting laws and propositions are presented. Based on this framework, we show how to model-check some interesting problems, e.g. are the required channel variables available before executing an interaction? is an unexpected state reachable or not? Furthermore, by some examples, we show how to describe business protocol in CDL.
and translate CDL into Promela, the input language of the SPIN model-checker, for automatically analyzing Linear Temporal Logic (LTL) properties.

The paper is organized as follows. We first introduce a choreography language CDL with formal syntax and semantics in Section 2. Section 3 describes two business protocols in CDL. Section 4 shows how to translate a choreography specification expressed in CDL to Promela for checking LTL properties, using the SPIN tool. Section 5 discusses related work, and Section 6 concludes.

2 CDL: A Choreography Description Language

In this section we define a small language CDL, which models choreography with a set of participant roles and the collaboration among them. Since the participants in a choreography always play some roles for the cooperation, we simply use the term role instead of the longer word “participant” in the rest of the paper.

2.1 Syntax

In language CDL, we have two kinds of variables, namely, state variables and channel variables, where the state variables keep track of the state of a role, and the channel variables can record channel instances used in the communications. Here each variable belongs to a determined role, while even with the same name, any two variables of different roles have nothing to do with each other.

In the following definitions, we let that meta-variable \( C \) ranges over names of choreography; \( R \) ranges over role declarations; \( A, B, A_1, A_2 \) etc. range over activity declarations; \( r, r_1 \) and \( r_2 \) range over role names; \( ch \) ranges over channel variable names; \( x \) and \( y \) range over variable names, which can be either state or channel variable names; \( op \) ranges over operation name; \( e, e_1 \) and \( e_2 \) ranges over expressions; \( g_i (i \in I) \), \( g \) and \( p \) range over boolean expressions, where \( I \) is a finite non-empty subset of natural numbers.

We use \( R \) as a shorthand for \( R_1, \cdots, R_n \), for some number \( n \). Similarly, for \( \bar{x}, \bar{op}, \bar{e} \), etc. We use \( r.x \) to refer to the variable \( x \) in role \( r \).

A choreography declaration consists of a name \( C \), some participant roles \( \overline{R} \), an activity \( A \), and a set of variable initializations \( \Sigma \) that assigns each variable with an initial value. A choreography (specification) takes the form:

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

Each role declaration consists of a name \( r \), some local variables \( \bar{x} \) and some observable behaviors represented as a set of operations \( \overline{op} \). The signature and function of the operations are not modeled in this work, that means, we take the operations here only as a set of names. A role with name \( r \) is defined as:

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

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.

Here is the syntax of basic activities:

\[
BA ::= \begin{align*}
&\text{skip} \quad \text{(skip)} \\
&| r.x := e \quad \text{(assign)} \\
&| (r_1 \to r_2, ch, rec, op) \quad \text{(request)} \\
&| (r_1 \leftarrow r_2, ch, rec, op) \quad \text{(response)}
\end{align*}
\]

\[
e ::= \text{null} | r.x \mid xp \quad \text{(expression)}
\]

The basic activities include skip, assignment and interaction. The skip activity does nothing. The assignment activity \(r.x := e\) assigns, within the role \(r\), the value of expression \(e\) to the variable \(x\). Here the expression \(e\) is either of the form \(\text{null}\) which denotes a special channel value of channel variable, or an XPath expression \(xp\), or a variable \(r.x\). We omit the details of the XPath expressions in this work, and assume that they denote some basic values (for example, boolean value true and false) or channels (channel instances). For \(r.x := e\), any free variable of \(e\) must belong to role \(r\), that is, an assignment is local to a specific role.

The most complex form of basic activities is interaction. An interaction activity is either a request or a response activity, in which operation \(op\) specifies what the recipient should do when it receives the message. The channel variable \(ch\) specifies where the information is sent to in the interaction. The \(rec\) denotes a list of variable recordings that capture observable information changes happened as the result of the interaction, with the form \(r_1.x := \pi_1; r_2.y := \pi_2\), where \(\pi\) and \(\gamma\) are two lists of variables on the roles \(r_1\) and \(r_2\) respectively. Here we don’t model the information exchange that occur during an interaction, but only care about variable recordings that capture observable information changes that can occur as a result of carrying out the interaction.

The control-flow activity is either a sequence activity \(A;B\), a non-deterministic activity \(A \sqcap B\), a parallel activity \(A \parallel B\), or a guarded choice \([i \in I \ g_i \Rightarrow A_i]\), where \(I\) is a finite non-empty subset of natural numbers.

\[
A, B ::= \begin{align*}
&BA \quad \text{(basic)} \\
&| p?A \quad \text{(condition)} \\
&| p*A \quad \text{(repeat)} \\
&| g:A:p \quad \text{(workunit)} \\
&| A;B \quad \text{(sequence)} \\
&| A \sqcap B \quad \text{(non-deterministic)} \\
&| A \parallel B \quad \text{(parallel)} \\
&| [i \in I \ g_i \Rightarrow A_i] \quad \text{(guarded choice)}
\end{align*}
\]

In the execution guarded choice is blocked for the guard of at least one branch to become true, then turns to the corresponding activity. The details of this will be defined below. A guard \(g\) is a boolean expression with the form:

\[
g_i ::= r.x < e \mid r.x = e \mid g_i \land g_i \mid \neg g_i
\]
If a guarded choice has only one branch, we will abbreviate it to the form of \( g \Rightarrow A \), which means “blocked the execution until \( g \) becoming true then \( A \)”.

Clearly, not all choreographies corresponding to the CDL syntax are meaningful. For example, to assign a boolean value to a variable and then use the variable as a channel in communication makes no sense. We should have a typing system for checking the well-formedness of choreographies in CDL. However, this is tedious but not very hard, also it is not the focus of this work. We omit the formal treatment of this here, and list only some important well-formedness rules here:

1. In a choreography, different roles have different names.
2. In a role definition, different variables have different names, and different operations have different names.
3. Every variable used in a choreography must be defined and initialized.
4. Every variable must be assigned with values of proper type, e.g. channel variables can only be assigned with channel instances.

In the following, we consider only the well-formed CDL specifications.

2.2 Semantics

In this section, the operational semantics for CDL is presented. The semantics is given by transition rules between configurations.

A configuration is a tuple of the form \( \langle A, \Delta \rangle \), where \( A \) is an activity, and \( \Delta \) is a state of the choreography under consideration, which is a function from variable names of all the roles to their values. In representing the state, each variable name is still decorated with the role name on which it resides, e.g., “r.x” represents a variable named \( x \) on role \( r \). The special value \( \text{null} \) for a channel variable on role \( r \) means that \( r \) does not know this channel. The initial state of a choreography can be straightly obtained from the variable initializations \( \Sigma \) part of the choreography declaration.

In the semantic rules below, we use notation \( \Delta[\sigma/r.] \) to denote the global state obtained from \( \Delta \) by giving values for variables \( \sigma \) to \( r \) while the values of other variables are unchanged, and use \( \Delta[\sigma/r.x] \) to denote the global state obtained from \( \Delta \) by giving new values for some variables on one or more roles. Moreover, we use \( \langle \epsilon, \Delta \rangle \) to denote the terminal configuration with empty activity text.

**Basic Activity.** The semantics of the basic activities is defined as follows:

Activity \( \text{skip} \) will always terminate successfully, and leave everything unchanged.

\[
\langle \text{skip}, \Delta \rangle \rightarrow \langle \epsilon, \Delta \rangle \quad \text{(SKIP)}
\]

The assignment activity updates variable \( r.x \) with the value of expression \( e \). Here we use \( \Delta \models e \downarrow v \) to mean that the expression \( e \) evaluates to \( v \) under state \( \Delta \):

\[
\Delta \models e \downarrow v
\]

\[
\langle r.x := e, \Delta \rangle \rightarrow \langle \epsilon, \Delta[v/r.x] \rangle \quad \text{(ASN)}
\]

Because we have omitted the detailed syntax of expressions, we will not consider the details of evaluation of expressions here.
An interaction is executed only if the value of the dedicated channel variable \(ch\) is not \textit{null}. After the interaction, there may be some variable updates on both roles. Here we always assume the atomicity of communication. The semantic rules for communication are given below where we assume that \(rec\) is \(r_1.x := \overline{v_1}; r_2.y := \overline{v_2}\).

\[
\Delta(r_1.ch) \neq \text{null}, \quad \Delta \models \overline{v_1} \downarrow \overline{v_1}, \quad \Delta \models \overline{v_2} \downarrow \overline{v_2},
\]

\[
\langle (r_1 \rightarrow r_2, ch, rec, op), \Delta \rangle \quad \rightarrow \quad (\epsilon, \Delta [\overline{v_1}/r_1.x, \overline{v_2}/r_2.y, r_2.ch])\quad \tag{REQ}
\]

\[
\Delta(r_2.ch) \neq \text{null}, \quad \Delta \models \overline{v_1} \downarrow \overline{v_1}, \quad \Delta \models \overline{v_2} \downarrow \overline{v_2},
\]

\[
\langle (r_1 \leftarrow r_2, ch, rec, op), \Delta \rangle \quad \rightarrow \quad (\epsilon, \Delta [\overline{v_1}/r_1.x, \overline{v_2}/r_2.y])\quad \tag{RES}
\]

Clearly, for the interaction to be possible, the message sender, i.e., \(r_1\) in rule (REQ) and \(r_2\) in rule (RES), must know the channel used in the communication.

\textbf{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.

\[
\Delta \models p \rightarrow \text{false}
\]

\[
\langle p?A, \Delta \rangle \quad \rightarrow \quad (\epsilon, \Delta)\quad \tag{IF-FALSE}
\]

\[
\Delta \models p \rightarrow \text{true}
\]

\[
\langle p?A, \Delta \rangle \quad \rightarrow \quad (A, \Delta)\quad \tag{IF-TRUE}
\]

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

\[
\Delta \models p \rightarrow \text{false}
\]

\[
\langle p \ast A, \Delta \rangle \quad \rightarrow \quad (\epsilon, \Delta)\quad \tag{REP-FALSE}
\]

\[
\Delta \models p \rightarrow \text{true}
\]

\[
\langle p \ast A, \Delta \rangle \quad \rightarrow \quad (A; p \ast A, \Delta)\quad \tag{REP-TRUE}
\]

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.

\[
\Delta \models g \rightarrow \text{true}
\]

\[
\langle g:A;p, \Delta \rangle \quad \rightarrow \quad (A; p^!(g:A;p), \Delta)\quad \tag{BLOCK}
\]

\textbf{Control-flow Activity.} The sequential composition \(A; B\) first behaves like \(A\). When activity \(A\) terminates successfully, it continues by behaving like \(B\). If \(A\) never terminates successfully, neither does \(A; B\).

\[
\langle A, \Delta \rangle \quad \rightarrow \quad (A', \Delta')\quad \tag{SEQ}
\]

\[
\langle A; B, \Delta \rangle \quad \rightarrow \quad (A'; B, \Delta')\quad \tag{SEQ-ELIM}
\]

\[
\langle \epsilon; B, \Delta \rangle \quad \rightarrow \quad (B, \Delta)\quad \tag{SEQ-ELIM}
\]

6
The non-deterministic choice $A \cap B$ behaves like either $A$ or $B$, where the selection between these branches is non-deterministic and internal, without referring the knowledge or control of the environment.

\[
\langle A \cap B, \Delta \rangle \rightarrow \langle A, \Delta \rangle \quad \text{(NON-DET1)}
\]
\[
\langle A \cap B, \Delta \rangle \rightarrow \langle B, \Delta \rangle \quad \text{(NON-DET2)}
\]

The guarded choice $\bigwedge_{i \in I} g_i \Rightarrow A_i$ behaves like $A_i$ if $g_i$ is the first guard by the textual order that evaluates to boolean value $true$ in state $\Delta$.

\[
\Delta \models g_i \rightarrow true, \Delta \models g_j \rightarrow false, j \in I, j < i \quad \langle \bigwedge_{i \in I} g_i \Rightarrow A_i, \Delta \rangle \rightarrow \langle A_i, \Delta \rangle \quad \text{(CHOICE)}
\]

Please note that, this rule implies that the activity will be blocked until some guard evaluates to $true$.

We use interleaving semantics for the parallel composition:

\[
\langle A, \Delta \rangle \rightarrow \langle A', \Delta' \rangle \quad \text{(PARA)}
\]
\[
\langle A \parallel B, \Delta \rangle \rightarrow \langle A' \parallel B, \Delta' \rangle \quad \text{(PARA)}
\]
\[
\langle B, \Delta \rangle \rightarrow \langle B', \Delta' \rangle \quad \text{(PARA)}
\]
\[
\langle \epsilon \parallel B, \Delta \rangle \rightarrow \langle B, \Delta \rangle \quad \text{(PARA-ELIM)}
\]
\[
\langle A \parallel \epsilon, \Delta \rangle \rightarrow \langle A, \Delta \rangle \quad \text{(PARA-ELIM)}
\]

Please note that, in CDL, we do not have communication between parallel branches, as the case in WS-CDL. On the other hand, because the existence of guard (guarded choice), we can have synchronization between the parallel branches, that is, one branch may wait for some other parallel branch(es) to make its guard(s) becoming true.

## 3 Modeling Choreography with CDL

In this section we illustrate with practical examples to show how the language CDL can be used to model Web services choreography specification.

### 3.1 A T-Shirts Procurement Protocol

The example in Figure 1 is adopted from Kavantzas’s use-case [16], which describes a protocol for purchase orders between a really big corporation (RBC) and a small T-shirts company (STC). In [9], this protocol is described using two (local) state variables, \texttt{AbortRequested} at role STC and \texttt{ConfArrived} at role RBC, both initialized to be \texttt{false}.

The informal description of the protocol as follows.
Informal Description. Here we give an informal description in the first:

- RBC sends a purchase order (PO) to STC.
- STC acknowledges the PO and initiates a business process to handle the PO.
- At this stage the interactions are divided into the parallel composition of two behaviours. In one thread of interaction, we have:
  - STC will, at some point, check AbortRequested is true (i.e. RBC’s abort request has arrived) or false (i.e. RBC’s abort request has not arrived).
  - If AbortRequested is false, then STC will send a PO confirmation message. When RBC receives it, it will set its ConfArrived to be true, and STC moves to the completion of PO processing.
  - If AbortRequested is true, then STC will send a AbortConfirmed message. RBC receives it, and in both sites the PO process aborts.

In another thread of interaction, we have:

- At some point RBC will check ConfArrived.
- If it is false (i.e. a PO confirmation has not arrived), then sends AbortRequest message to STC.
- If it is true (i.e. a PO confirmation has arrived), then RBC moves to the completion of PO processing.

Representation in CDL. Now we give the formal description of above protocol using our language CDL. In the following, we will use names $R$ and $S$ to denote two roles $RBC$ and $STC$ respectively, and use $ch_r$ and $ch_s$ for channel variables referring to corresponding roles $R$ and $S$. For convenience, we attach labels $I_i$ ($i = 0, \cdots, 4$) to each of the interactions in the protocol.

Here are the role declarations of the protocol:

$$R \{[ch_r, ch_s, ConfArrived], \{OrderAck, POConfirmation, ConfirmAbort\}\}$$

$$S \{[ch_r, ch_s, AbortRequested], \{CreatOrder, Abort\}\}$$
Based on the protocol, $\Sigma$ takes the set of variable initializations as follows:

\[
\{R.\text{ConfArrived} := \text{false}, R.ch := ULR_r, R.chs := URL_s, \\
S.\text{AbortRequested} := \text{false}, S.ch := URL_r, S.chs := URL_s\}
\]

Where $URL_r$ and $URL_s$ are the address information of roles $R$ and $S$ respectively.

The activity $A$ of the choreography are defined as follows:

\[
A = I_0; I_1; (A_1 \parallel A_2)
\]

\[
I_0 = (R \rightarrow S, ch_s, \{\}, \text{CreateOrder})
I_1 = (S \rightarrow R, ch_r, \{\}, \text{OrderAck})
A_1 = (S.\text{AbortRequested} = \text{false}) \Rightarrow I_2 \parallel (S.\text{AbortRequested} = \text{true}) \Rightarrow I_3
A_2 = (R.\text{ConfArrived} = \text{false}) \Rightarrow I_4
I_2 = (S \rightarrow R, ch_r, \{R.\text{ConfArrived} := \text{true}\}, \text{POConfirmation})
I_3 = (S \rightarrow R, ch_r, \{\}, \text{ConfirmAbort})
I_4 = (R \rightarrow S, ch_s, \{S.\text{AbortRequested} := \text{true}\}, \text{Abort})
\]

This completes a formal representation of the protocol in CDL, which further can be based on to analyze the behavior of the protocol in Section 4.1.

### 3.2 Dynamic Routing Protocol

The protocol presented in this section comes from the dynamic routing example of service interaction patterns in [4]. We have given an algorithm in [23] to check if the protocol will be stuck due to the fact that some required channel is not available (i.e. channel-absence) when an interaction is executed. Here we model this protocol in CDL which will be used as also an example in Section 4.2 for checking the channel absence using SPIN tool.

Informal Description. In Figure 2, there are six roles involved in this protocol: Buyer (B), Sales department (S), Finance (F), Warehouse (W), Shipper nominated by the buyer (SH_b), and the default Shipper (SH_w) known by the warehouse. The protocol describes a process in which the buyer makes a purchase order for a product to the sales department. After processing the received order, the sales department sends a request to the finance department to process to generate an invoice and payment receipt for the order. This request contains a reference to the buyer’s procurement service and possibly also to a shipping service nominated by the buyer. After arranging invoicing and payment by interacting directly with the buyer, the finance service forwards the order to the warehouse service. The warehouse issues a request to a shipping service which may be either the company’s default shipping service, or the one originally nominated by the buyer. The shipping service eventually sends a shipping notification directly to the buyer.
Representatiom in CDL. Now we give the formal description of the above protocol using CDL. In the following, we use names B, S, F, W, SHb and SHw to denote the six roles respectively, and use \( ch_b, ch_s, ch_f, ch_w, ch_{sb} \) and \( ch_{sw} \) for channel variables referring to corresponding roles. Particularly, \( ch_{sb} \) refers to the address of the shipper \( SH_b \) nominated by the buyer, and \( ch_{sw} \) refers to the address of the default shipper known by warehouse.

Here are the role declarations of the protocol.

\[
\begin{align*}
B \ &= \ {[ch_b, ch_s, ch_{sb}], \{payment\}} \\
S \ &= \ {[ch_s, ch_f, ch_b], \{poReq\}} \\
F \ &= \ {[ch_f, ch_w, ch_b], \{payReq\}} \\
W \ &= \ {[ch_w, ch_{sw}, ch_b], \{pickReq\}} \\
SH_w \ &= \ {[ch_{sw}, ch_b], \{shipReq, notify\}} \\
SH_b \ &= \ {[ch_{sb}, ch_b], \{shipReq, notify\}}
\end{align*}
\]

Based on the protocol, the variable initialization \( \Sigma \) is defined as follows:

\[
\begin{align*}
\{B.ch_b := URL_b, \ B.ch_s := URL_s, \ B.ch_{sb} := URL_{sb}, \\
S.ch_w := URL_w, \ S.ch_f := URL_f, \ S.ch_b := null, \\
F.ch_f := URL_f, \ F.ch_w := URL_w, \ F.ch_b := null, \\
W.ch_w := URL_w, \ W.ch_{sw} := URL_{sw}, \ W.ch_b := null, \\
SH_w.ch_{sw} := URL_{sw}, \ SH_w.ch_b := null, \\
SH_b.ch_{sb} := URL_{sb}, \ SH_b.ch_b := null\}
\end{align*}
\]

Where initially, all the roles know the addresses (i.e. channel instances) of themselves. Moreover, the buyer knows the address \( URL_{sb} \) of its nominated shipper, the
sales department knows the address URL of finance department that knows the address URL of warehouse, and warehouse knows the address URL of its default shipper.

Following the informal description of the protocol, we (might) write down the activity $A$ of the choreography as follows:

$I_0 :: (B \rightarrow S, ch_s, \{S.ch_b := B.ch_b\}, poReq);$
$I_1 :: (S \rightarrow F, ch_f, \{F.ch_b := S.ch_b\}, payReq);$
$I_2 :: (F \rightarrow B, ch_b, \{\}, payment);$
$I_3 :: (F \rightarrow W, ch_w, \{W.ch_b := F.ch_b\}, pickReq);$
   \{I_4 :: (W \rightarrow SH_b, ch_{sb}, \{SH_b.ch_b := W.ch_b\}, shipReq);$
$I_5 :: (B \leftarrow SH_b, ch_b, \{\}, notify) \}$
$I_6 :: (W \rightarrow SH_w, ch_{sw}, \{SH_w.ch_b := W.ch_b\}, shipReq);$
$I_7 :: (B \leftarrow SH_w, ch_b, \{\}, notify) \}$

This completes a formal representation of the protocol in CDL. It is not easy to see whether some channel passing-related defects exist in this choreography.

4 Checking Choreography Using SPIN

In this section we discuss how to verify a given choreography specification using the SPIN model-checker [15]. 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) linear temporal logic properties of Promela specifications using an exhaustive state space search.

4.1 The T-Shirts Procurement Protocol

Translation to Promela. We illustrate our translation procedure based on the purchase order example in Section 3.1. We list the translated Promela code of this example in Figure 3 and Figure 4. From the code and following explanation, we can see the basic translation procedure here.

The first part of the code, as shown in Figure 3, consists of some type declarations and variable declarations. We introduce a variable named $r_x$ for variable $x$ under role $r$ in the choreography.

It is a critical problem to avoid state explosion in model-checking. If variables have a wide range of possible values, e.g. integers, the performance of model-checking will be quite poor. In our model, we only consider the values of state variables and channel variables, which can be easily enumerated in the initial $mtype$ declaration; while the information variables in WS-CDL are not considered. Therefore, the size of the state space is guaranteed to be small. We also use $null$ to represent the value of channel variable in the case that the channel variable does not contain any address information.
mtype = {null, True, False, RBC, STC, URLr, URLs, FOConfirmation, ConfirmAbort, Abort, CreateOrder, OrderAck, I0, I1, I2, I3, I4};

#define intr(name_, from_, to_, channel_, op_) \
  atomic { \
    assert channel_ != null; \
    name = name_; \
    from = from_; \
    to = to_; \
    channel = channel_; \
    op = op_; \
  }

mtype RBC_ConfArrived; mtype RBC_chs; mtype RBC_chr; mtype STC_AbortRequested; mtype STC_chs; mtype STC_chr;

mtype name = null; mtype from = null; mtype to = null; mtype channel = null; mtype op = null;
bool para_aux_1 = false; bool para_aux_2 = false;

Figure 3. Promela Code: Declarations

To help describe temporal properties, we introduce some snapshot variables such as name, from and to to keep track of the current interaction. Each interaction of the choreography is translated as a macro intr, where we test if the corresponding channel is initialized, and update these variables. We augment the channel variable ch into the form from_ch. The variable recordings are translated as assignment statements after the corresponding interaction. We use atomic to make sure each interaction is an atomic step. We also introduce some auxiliary boolean variables to implement parallelism, which will be discussed soon.

The code in Figure 4 consists of several processes that denotes the choreography body. The init process initializes the variables on each role, and starts the chor process. The chor process do some interactions first, and then start two parallel processes to implement the parallel composition in the example. The if statement can be used to implement both kinds of choice structures proposed in our formal model. 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. However, in the WS-CDL specification, the first branch is selected when multiple guards are true. Thus we modify the guard for the ith branch into the form gi ∧ ¬g1 ∧ ··· ∧ ¬gi−1.

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. For parallel activities, we first introduce some auxiliary processes with the prefix “par” for each block in the parallel activity, and then call the processes to start by a run statement. We use conditional expressions such as r_para_aux_i == true to block corresponding run statements. The auxiliary variables such as r_para_aux_i are assigned by true only at the end of each called process, thus achieving the synchronous calling mechanism. Although we can actually omit this detail in this example because we don’t have
proctype par1() {
  if :: STC_AbortRequested == False -> atomic {
    intr(I2, STC, RBC, STC_chr, POConfirmation); RBC_ConfArrived = True;
    :: STC_AbortRequested == True ->
    intr(I3, STC, RBC, STC_chr, ConfirmAbort);
    fi;
    para_aux_1 = true;
  }
}

proctype par2() {
  if :: RBC_ConfArrived == False -> atomic {
    intr(I4, RBC, STC, RBC_chs, Abort); STC_AbortRequested = True;
    fi;
    para_aux_2 = true;
  }
}

proctype chor() {
  intr(I0, RBC, STC, RBC_chs, CreateOrder);
  intr(I1, STC, RBC, STC_chr, OrderAck);
  run par1(); run par2();
  para_aux_1 == true; para_aux_2 == true;
}

init {
  atomic {
    RBC_chr = URLr; RBC_chs = URLs; STC_chr = URLr; STC_chs = URLs;
    RBC_ConfArrived = False; STC_AbortRequested = False;
  }
  run chor();
}

Figure 4. Promela Code: Processes

any other interaction after the parallel composition, the above treatment is necessary for translating a general choreography into Promela.

In Table 1 we give a mapping from CDL to Promela code. With these translation rules, it is not hard to implement an automatic translation tool. Since Promela supports most of the activities defined in our semantics, most translation is quite straightforward.

Verification. Based on the translated Promela code, We have checked two LTL properties with SPIN. These properties are taken from [9].

– The protocol never moves to the situation where STC sends a PO confirmation but RBC aborts:

! (<> (to==RBC & op==POConfirmation) &
  <> (to==RBC & op==ConfirmAbort))

As expected, SPIN reported that the above property holds.

– It is possible that STC may receive AbortRequest message, and STC still sends a PO-confirmation message to RBC.

13
mtype = {null, True, False, B, S, F, SH, URLb, URLs, URLf, URLw, URLsw, URLsb, payReq, pickReq, shipReq, notify, I0, I1, I2, I3, I4, I5, I6, I7};
#define intr ... 
mtype B_chb; mtype B_chs; mtype B_chsb; ...

mtype name = null; mtype from = null; mtype to = null; ...

init {
  atomic {
    B_chb = URLb; B_chs = URLs; B_chsb = URLsb; ...
  }
  run chor();
}

proctype chor() {
  atomic {
    intr(I0, B, S, B_chs, poReq); S_chb = B_chb;
  }
  atomic {
    intr(I1, S, F, S_chf, payReq); F_chb = S_chb;
  }
  intr(I2, F, B, F_chb, payment);
  atomic {
    intr(I3, F, W, F_chw, pickReq); W_chb = F_chb;
  }
  if :: true ->
    atomic {
      intr(I4, W, SH, W_chsw, shipReq); SHb_chb = W_chb;
    }
  intr(I5, SHb, B, SHb_chb, notify);
  :: true ->
  atomic {
    intr(I6, W, SHw, W_chsw, shipReq); SHw_chb = W_chb;
  }
  intr(I7, SHw, B, SHw_chb, notify);
  fi
}

Figure 5. Promela Code for the Dynamic Routing Protocol

<> (STC_AbortRequested==True) &&
<> (from==STC && to==RBC && op==POConfirmation)

This property is an existence property. SPIN gives a path on which the property holds as a counter example. Thus we know the property holds, too.

4.2 The Dynamic Routing Protocol

We can similarly translate the dynamic routing protocol into Promela. Since the translated code is similar to the purchase order example, we omit some of the tedious declaration and initialization codes here. Figure 5 illustrates the code for the choreography. Most of the code are quite straightforward. To implement the non-deterministic choice, we use an if statement with guards set to true.

Using SPIN, we can check if the choreography will be stuck due to a required channel is not available to a participant when an interaction needs to be carried out.

[] (! timeout)

Actually, the above property can be used to check arbitrary deadlock problems. The designer can further understand the cause of the deadlock by studying the counter example provided by SPIN. Concretely in the choreography defined above, we can find that the interactions $I_1, I_2$ and
interaction $I_3$ are all fine. However, interaction $I_4$ is not executable because when the execution arrives to
the point where interaction $I_3$ completes, role $W$ (the warehouse) does not know the channel $ch_{sb}$ of nominated shipper; thus the execution gets stuck. To resolve this problem, we can pass the address $URL_{sb}$ from role $B$ to role $S$, then from $S$ to $F$, finally from $F$ to $W$.

5 Related Work

In recent years, many researches pay attention to the study of the formal foundation of choreogra-
phy based on process calculi. Brogi et al. [6] presented a formalization of Web Service Choreogra-
phy Interface (WSCI) using CCS [19], and discussed the benefits of such formalization. In [8], Busi et al. proposed a simple choreography language whose main concepts are based on WS-
CDL. Foster et al. [12] discussed a model-based approach to verify Web services compositions.
In [24], Zaha et al. presented a language Let’s Dance for modeling service choreography tar-
geting the early phases of the development life cycle. The language Let’s dance main focuses
on the control flow aspect of choreography, and simply denotes elementary interaction as atomic.
Li et al. studied the semantics of WS-CDL [17] and verified Web services choreography using
process algebra [18].

There are some literatures on the modeling of interactions in choreography. In [14], Gorrieri
et al. presented formal semantics of a significant fragment of WS-CDL that provides a mean to
deal with interactions, and reasoned about the adequacy of such interaction patterns when the
alignment property is considered. In [9], Carbone et al. defined a “global calculus” originated
from WS-CDL based on the session types. A session is initiated by a service channel with fresh
session channels and interactions. An interaction is the in-session communication over a session
channel.

For pushing the service composition technology to progress further, Barros [4] presented a
collection of patterns of service interactions that allow emerging Web services functionality to be
benchmarked against abstracted forms of representative scenarios. The dynamic routing protocol
presented in this paper demonstrates one of these patterns. Moreover, Decker [10] represented
several of these service interaction patterns by using $\pi$-calculus.

As for the projection and conformance validation between choreography and orchestration,
much work has been carried out, while much is still on going. Carbone et al. [9] studied the
description of communication behaviors from both global message flows and end-point behav-
ior levels respectively. Three principles for well-structured global description and a theory for
projection are developed. In [7], Busi et al. formalized conformance with a bisimulation-like
relation. By means of automata, Schifanella et al. [3] defined a conformance notion that tests
whether interoperability is guaranteed. Fu et al. [13] specified a conversation protocol by a re-
alisable Büchi automaton, and the peer implementations are synthesized from the protocol via
They defined an effective procedure that can be used to verify whether a service with a given
contract can correctly play a specific role within a choreography. Moreover, Decker et al. dis-
cussed the issue of local enforceability of Let's Dance choreographies in [11]. van der Aalst [22]
focused on conformance by comparing the observed behavior recorded in logs with some pre-
deﬁned model. In [21], Qiu et al. deﬁned the concept of restricted natural choreography that
is easily implementable, and proposed two structural conditions as a criterion to distinguish the
restricted natural choreography.
6 Conclusion

With the blooming of Web technology, more and more computation are established by Web services residing over the Internet. For accomplishing the goal of the computation, they should not only have “correct” functionalities, but also correct interactions with each other. With the interaction becoming more complex, the problems related to specify and verify the interaction of the participants will become harder, too.

The goal in the designing of choreography description language CDL is to provide a concise formal model of choreography, while still characterizing the key features of Web services choreography. Within this work, we meet many problems with the choreography language definition, and uncover some problems which are not clearly (or not adequately) defined in the WS-CDL specification. For instance, WS-CDL specification has no explicit definition for channel variable initialization, which is important for judging if a required channel variable is available or not. Based on the model, it is also possible to verify many interesting properties of a given choreography.

The main contribution of this paper are:

– Besides normal choreography concepts, we model the interaction with recordings of state and channel variable changes, which provide the capability for verifying data-related properties such as availability of channel variables and reachability of states.

– We provide a set of translation rules from CDL to Promela, which allows the user to model-check choreographies in SPIN.

Compared with WS-CDL, CDL is still a subset of WS-CDL. We have not covered concepts such as exception handling and finalization, which are possible future work. Also, for model-checking a choreography in WS-CDL, we still need to develop a tool for translating from WS-CDL to CDL, which can be further translated to Promela for model-checking. The development of both tools are our future work.

References

<table>
<thead>
<tr>
<th>expression</th>
<th>definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>skip</td>
<td>skip</td>
</tr>
<tr>
<td><code>r.x := e</code></td>
<td><code>r.x = e</code></td>
</tr>
</tbody>
</table>
| `(r_1 → r_2, ch, rec, op)` | atomic{
  intr(I, r_1, r_2, r_1_ch, op);
  r_1_x = e_1; ...  
  r_2_y = e_2; ...  
  }
| `(r_1 ← r_2, ch, rec, op)` | atomic{
  intr(I, r_2, r_1, r_2_ch, op);
  r_1_x = e_1; ...  
  r_2_y = e_2; ...  
  }
| `A; B`     | `A; B`     |
| `p?A`      | if
  :
  :: p -> A
  :: !p-> skip
  fi
| `\{ \}_{i∈I} g_i ⇒ A_i` | if
  :: g_1 → A_1
  :: g_2 && !g_1 → A_2
  :: g_3 && !g_1 && !g_2 → A_3
  ...  
  fi
| `A \cap B` | if
  :: true → A
  :: true → B
  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`    | atomic {
  run r_parA();
  run r_parB();
  
  r_para_aux_A == true;
  r_para_aux_B == true;
  }

where `I` is the name of the interaction, and `rec` is `r_1 := r_1'; r_2 := r_2'`