An Approach to Checking Choreography with Channel Passing in WS-CDL

Cai Chao Qiu Zongyan
LMAM & Department of Informatics, School of Math.
Peking University, Beijing, CHINA
{caic,qzy}@math.pku.edu.cn

Abstract WS-CDL is a language to describe Web services choreographies, which can be projected into individual services or orchestration skeletons. This language adopts a channel passing mechanism to support dynamic Web services composition. It is a challenge to ensure the services generated from a choreography always have access to sufficient and correct channels to complete their collaboration. In fact, WS-CDL is not ready for rigorous validating and implementing with respect to channel passing, since it implicitly assumes which role should firstly initialize a channel variable. In this paper, we propose an algorithm to uncover these implicit assumptions. The algorithm is implemented as an extension to Pi4SOA. With the results of the algorithm, some existing methods for verification and implementation can be applied on choreographies written in WS-CDL. In addition, we propose an approach to detect design defects in choreographies. Using the approach, a defect is found in a sample choreography from the WS-CDL Primer. It seems that choreographies with channel passing are error prone. Methods and tools are necessary to support designers in this field. Also, we suggest improving the situation by adding a syntactical construct to WS-CDL.

1 Web Service Composition and Channel Passing

Web service composition refers to the process of combining several web services to provide a value-added service. It is fundamental to the support of the web-based business applications. Two levels of views to the composition of web services exist, namely orchestration and choreography. The orchestration view focuses on interactions between one party and the others, while the choreography view emphasizes a global perspective.

In the large service-oriented systems, stockholders may require a global picture of the way by which services interact with each other, rather than multiple small pictures of individual services [16]. Developed by W3C’s Choreography Working Group, WS-CDL [3] describes business protocols from a global viewpoint and the description will be implemented by individual distributed processes without center control [8].

* Supported by National Natural Science Foundation of China (No. 60773161)
WS-CDL is developed to enable a kind of top-down design approach. The developers of a business process may use a pre-developed WS-CDL document (choreography) as a guide to direct their work, or to generate a behavioral interface in the form of an abstract WS-BPEL definition that describes the activities and conditions for each participants in a choreography [6,5].

The Web Services Choreography working group believes that, with WS-CDL, the cost of implementing Web services may be reduced by ensuring conformance to expected behavior, and/or increase the utility of Web services as they will be able to be shown to meet contractual behavior [5]. Furthermore, the verification and validation may be done on choreography, so that the services which conform to the choreography can form a well-behaved system.

As for the modeling and projection from choreography to orchestration, some works have been carried out. Fu et al. [10] specified a conversation protocol by a realizable Büchi automaton, and the participant implementations are synthesized from the protocol via projection. Carbone et al. [9] studied the description of communication behaviors from both global message flows and end-point behavior levels respectively. Our previous work [12] proposed some novel language features for specifying choreography to enhance the top-down approach. All these works have a conclusion that if a choreography satisfies some criteria, we can automatically get a group of implementations whose collective behavior is equivalent to the original choreography.

All these works show that choreography is a helpful layer (tool/language) for designing and implementing web services. Verification is always easier on the choreography level than on the orchestration level, and the correctness of services generated from a verified choreography may be guaranteed. However, none of these work cover the channel passing mechanism, although it is a fundamental feature in WS-CDL for supporting of dynamic service composition. Taking this feature into the work, we must consider and reconsider many new and old problems.

1.1 Channels and Channel-Passing

Web services interact with each other through channels (or channel instances). When all channels are known by their users and all usage of the channels are described statically in the specification of the process, we say that the process has a static communication structure. Processes of this catalogue are relatively simple, which have been studied for a long time.

In the Web world, the static communication structure is often not sufficient. A business process may involve several participants, where some of them may be selected dynamically during the run by some participant which has already taken into the work. Also, a participant may join the process only after some specific events in the execution. The new joined participant must be equipped with some channels to communicate with the participants which are already in the work. From this brief discussion, we can conclude that, in general, channel passing is indispensable, and is a crucial mechanism in practice. For this reason,
both WS-BPEL [4] and WS-CDL offer some mechanisms to support dynamic communication structures and channel passing.

The example in Figure 1 describes an online shopping process given in the MSC (Message Sequence Chart) form. After the buyer and seller make a bargain, a shipper should be be selected by a party, for example, the seller. The shipper will then contact with the buyer to confirm where and when to transport the goods from the seller’s warehouse to the buyer. In this scenario, the shipper may be selected by the seller dynamically according to some factors, e.g., the location of the buyer, or the quality of goods, etc. Clearly, the chosen shipper has neither idea of the buyer been served nor the place to deliver before the selection. Thus the seller must give the buyer’s contact information to her or him. The information may be either a phone number, a fax number or an address. In the Web service environment, the information must contain the contacting channels. That is why we need “channel passing” here.

New problems appear in Web service composition along with the present of channel passing. We list some important points here:

- **Channel-absent**: A required channel may be not available to a participant when an interaction needs to be carried out, because either the participant needs to use the channel in the interaction, or needs to pass the channel to another participant. In this case, the whole work will be stuck.

- **Channel-redundant**: In the execution flow, some channels may be held or obtained by some participant are never used. This can be a potential security problem.

To facilitate the design methodology promoted by choreography and relevant verifications, designers and researchers must conquer the following problems:

- Design a choreography language with suitable features for specifying channel, channel passing and related information.
- Design an automatic mechanism to produce implementations from choreographies together with channel passing.
- Validate choreographies so that the generated implementations will never suffer from problems, especially, the problems related to channel passing.
1.2 Channel Passing in WS-BPEL

WS-BPEL [4] is the de facto standard language for Web service orchestration recently. It supports dynamic composition through channel passing. A π-calculus [14] based semantics for BPEL has been proposed [11], where channel passing is modeled as a kind of location mobility. In this subsection, we roughly explain the mechanism and emphasize the concept “initial channel set”.

The services with which a business process interacts are modeled as partner links in WS-BPEL. As defined in the specification, the syntax of partner links is:

```xml
<partnerLinks>
  <partnerLink name="NCName"
     partnerLinkType="QName"
     myRole="NCName"? partnerRole="NCName"?
     initializePartnerRole="yes|no"? />+
</partnerLinks>
```

Each `<partnerLink>` can be seen as a channel variable, which has a name and a declared type. To invoke a service or communicate with some service, the sender must know the actual address. WS-Addressing [2] defines the construct endpoint reference (EPR, i.e., channel instance) to provide an interoperable and transport-neutral way of encoding addressing information, that is, where and how to send the message. A channel variable must be initialized with a channel instance before it is used in communication.

In WS-BPEL, two ways to initialize a channel variable are permitted. On one hand, the channel variable can be bound with an instance statically, either via an assignment in the business process definition, or as part of the process deployment, or execution environment configuration. On the other hand, it is also possible to bind partner links dynamically. The service could receive an EPR from some partner and assign it to a channel variable.

The attribute `initializePartnerRole` specifies whether the `partnerLink` must be initialized by the service. If the attribute is “yes”, the service must provide an EPR to initialize the channel variable before it is first used.

The set of channel variables, which must be initialized by each participant, is an important part of a business contract. We will call it the initial channel set. Each participant must choose some services for its variables in this set, thus the services are introduced into a business conversation.

1.3 Channel Passing in WS-CDL

WS-CDL provides a different set of terminologies to describe communication and relevant entities. A WS-CDL document includes declarations of `ChannelType`, `RoleType` and `ChannelVariable`.

An abbreviated structure of a `ChannelType` element is given below. A channel type has a name, and a role type, which specifies the destination of information exchanged on the instance of this channel type. Zero or more `passing` elements
specify some channel types which can be passed through channel instances of this type.

```xml
<channelType name="NCName" ...>
  <roleType typeRef="QName" .../>
  <passing channel="QName" .../>

  ......
</channelType>
```

Channel passing is supported by interactions. An interaction is associated with a channel variable, and messages are transformed through a channel instance assigned to that variable. Channel variables can be part of a message. In fact, WS-CDL always involves channel types and channel variables, but never channel instance. So we will use channel short for channel variable when choreography is talked.

We have observed that WS-CDL lacks of mechanism to specify who initializes a channel before it is first used. As we have discussed in Section 1.2, a service can obtain an instance for a variable in two ways, either choosing one by itself, or getting one from a partner. Current CDL provides the later but says nothing about the former. In other words, WS-CDL has not a construct “initial channel set” to describe which channel variables a role type should initialize before the choreography starts.

Without the initial channel set, choreography is not as powerful as it is designed to be. There are two main problems:

1. Properties of a choreography related to channel passing, e.g., channel-absent, channel-redundant, etc., can not be automatically checked, because any checking algorithm can do its job only if initial channel sets of each role is explicitly given.
2. Choreography can not be used to generate WS-BPEL processes or specifications, because the later need the instruction of whether a process should initialize a channel variable.

To explore the approach to solve the problems related to validation of WS-CDL Choreography with respect to channel passing is the goal of this paper.

### 1.4 Contributions of This Work

The identification of the validation and implementation problems in current WS-CDL is the first contribution of our paper. We provide here an approach which can help people checking the channel passing and implementing in choreography developed in current WS-CDL.

Firstly, we give a language Chor, which is a subset of WS-CDL focusing on channel passing related features. Then its operational semantics is given. To cope with the lacking of initial channel sets in WS-CDL, we propose a roundabout approach to solve the channel-related verification problem.

Based on the formal model, we develop an algorithm which derives from a choreography the underlying assumptions or requirements of initial channel
sets for the participants. We develop a Java implementation of the algorithm as an extension for Pi4SOA [1], which takes as input XML format choreographies written in WS-CDL, and outputs initial channel sets for each role. With these initial channel sets, our previously presented algorithms [15] can be applied to verify channel related problems. Also, our existing projection [7] could be used to generate individual services.

Furthermore, we propose a new approach to check design defects in WS-CDL choreographies, which may not necessarily be the channel related problems. Using this approach, a defect is even found in a sample choreography which is used throughout the WS-CDL Primer [13]. Experience shows that channel passing is error prone in choreography. Methods and tools are necessary to support designers in this direction.

In the following of this paper, we will give the syntax and semantics of Chor in the next section. The initial channel set derivation algorithm is described in Section 3. Then, we talk about the approach to check design defects in choreographies and some experience in Section 4. In Section 5 we discuss the approach and suggest a syntactical construct to improve WS-CDL.

2 A Formal Model of Channel Passing in WS-CDL

In this section, we give a concise language Chor, which models channel passing features in WS-CDL. Firstly, we present the syntax of the language. Then, its operational semantics will be given.

2.1 Syntax of Chor

A WS-CDL document (choreography) includes a declarative part and a behavioral part. A basic element in the latter part is interaction. The syntax of interaction is given as below.

```xml
<interaction name="NCName"
    channelVariable="QName" ...>
    <participate fromRoleTypeRef="QName"
        toRoleTypeRef="QName" .../>
    <exchange name="NCName"
        channelType="QName"?...>
        <send variable="..." />
        <receive variable="..." />
    </exchange>*
...</interaction>
```

In an interaction, the fromRoleTypeRef and the toRoleTypeRef exchange a message through the channelVariable. The message could also be a variable of some channelType. The main concepts here are roleType and channelVariable, which
we will mention frequently in this paper. We model them as “role” and “channel” in Chor_c. Each role has a unique name that ranges over r, r_1, . . . . Channels that range over c, c_1, . . . .

Now, we give out the syntax of Chor_c. A choreography consists of a static description for a group of roles, and their collaborative behavior A, with the form:

\[
chor ::= [RDecl, A]
\]

A role declaration has the form of:

\[
RDecl ::= r[Cin]
\]

where r is a name and Cin a set of channels on which the role listens. In WS-CDL, a channel variable must be of some channel type, and a channel type is associated with a role type. So, each channel belongs to some role type. Cin contains all channels which belong to r.

Each role involved in collaboration will execute a series of activities, which can be divided into two kinds: local activities and interactions. A local activity, such as silent activity and assignment, is performed by a single role. We will use meta-variables a_i, a_{i+1}, . . . for local activities performed by r_i. An interaction from r_i to r_j through channel c_1 is represented as c_1 : r_i \xrightarrow{c_2} r_j, where c_2 is the passed channel which can be used by the receiver r_j in the communication afterward. Multiple channels are permitted to be transferred in one interaction, which can also be simulated by several interactions which transfer one channel each time. Since we care only about the channel passing, we will use c_1 : r_i \xrightarrow{} r_j to represent a communication where no channel is passed.

Sequential, parallel and choice composition structures are employed here. The behavior of a choreography is defined as an activity A.

\[
A ::= c_1 : r_i \xrightarrow{c_2} r_j \text{ interaction} \\
| a_i \quad \text{local activity} \\
| A_1 : A_2 \quad \text{sequence} \\
| A_1 \parallel A_2 \quad \text{parallel} \\
| A_1 \cap A_2 \quad \text{choice}
\]

2.2 Semantics

Now we give the operational semantics of Chor_c to illustrate how the channel passing mechanism works.

Before defining the run-time configuration, we introduce a binary predicate K(r, c), which means that role r knows channel c. We promote K to set of channels. If C = {c_1, . . . , c_k} is a set of channels, then K(r, C) \equiv K(r, c_1) \land . . . \land K(r, c_k). We use “σ” to denote a proposition constructed using K, true and \land, which represents the channel knowledge state (or simply state) for all participants.

A configuration of a choreography is of the form (σ, A), where A is an activity, σ is a state.
A local activity of any peer can always execute and it does not change the state.

\[(\sigma, a^i) \rightarrow (\sigma, \epsilon)\]  \hspace{1cm} \text{(Local Activity)}

A communication from \(r^i\) through \(c_1\) can happen only if the sender \(r^i\) knows the channel \(c_1\). Besides, if a channel is passed, the sender must know that channel as well. After the interaction, the receiver will know the passed channel too.

\[
\sigma \vdash K(r^i, \{c_1, c_2\})
\]

\[
(\sigma, c_1 : r^i \xrightarrow{c_2} r^j) \rightarrow (\sigma \land K(r^j, c_2), \epsilon)
\]  \hspace{1cm} \text{(Communication)}

The transition rules for sequential and choice compositions are regular.

\[
(\sigma, A) \rightarrow (\sigma', \epsilon)
\]

\[
(\sigma, A; A') \rightarrow (\sigma', A')
\]  \hspace{1cm} \text{(Sequence)}

\[
(\sigma, A_1 \cap A_2) \rightarrow (\sigma, A_1)
\]  \hspace{1cm} \text{(Choice-1)}

\[
(\sigma, A_1 \cap A_2) \rightarrow (\sigma, A_2)
\]  \hspace{1cm} \text{(Choice-2)}

Parallel composition is easily handled here. All parallel branches run independently, and then the resulting states are combined.

\[
(\sigma, A_1) \rightarrow (\sigma_1, \epsilon)
\]

\[
(\sigma, A_2) \rightarrow (\sigma_2, \epsilon)
\]  \hspace{1cm} \text{(Parallel)}

Due to the nature of parallel composition, a branch may complete its work before the other branches start. This is the strictest requirement with respect to channel passing. The rule reflects this consideration.

As mentioned earlier, since WS-CDL lacks of mechanism to specify initial channel sets for roles, no role can know any channel information of its partners in an initial configuration. We are under a similar situation now. It is impossible for the choreography to execute any interaction.

However, it is possible to derive initial channel sets using the semantics, which will be studied in the next section.

3 Derive Initial Channel Sets

Based on the formal model, we develop an algorithm to derive the initial channel sets.

3.1 Basic Idea

According to the operational semantics, in order to execute an interaction \(c_1 : r^i \xrightarrow{c_2} r^j\), the sender should know both \(c_1\) and \(c_2\). A role can obtain a channel either from partners by channel passing or from the initial knowledge. The basic idea of our algorithm is to trace the execution of all roles in a choreography, accumulate both channel sets they use and obtain in the execution. The channels, which haven’t been obtained before the first use by a role, must be included in the “initial channel set” for this role.
check \((\Sigma, A, A_c)\)

if \(\Sigma = \emptyset \lor A = \epsilon\) then return \(\Sigma\)

switch according to the form of \(A\) do

\(\text{case } A' \rightarrow \text{ return } \text{check } (\Sigma, A_c, \epsilon)\)

\(\text{case } c_1 : r' \xrightarrow{c_2} r''\)

\(\text{foreach } \sigma \in \Sigma \text{ do}\)

\(\text{if } c_1 \not\in (\sigma r').K \text{ then}\)

\((\sigma r').K \cup = \{c_1\}\)

\((\sigma r').I \cup = \{c_1\}\)

\(\text{if } c_2 \not\in (\sigma r').K \text{ then}\)

\((\sigma r').K \cup = \{c_2\}\)

\((\sigma r').I \cup = \{c_2\}\)

\(\text{end}\)

\(\text{return } \text{check}(\Sigma, A_c, \epsilon)\)

\(\text{case } A_1; A_2 \rightarrow \text{ return } \text{check}(\Sigma, A_1, A_2, A_c)\)

\(\text{case } A_1 \cap A_2:\)

\(\Sigma_1 := \text{check}(\Sigma, A_1, A_c)\)

\(\Sigma_2 := \text{check}(\Sigma, A_2, A_c)\)

\(\text{return } \Sigma_1 \cup \Sigma_2\)

\(\text{case } A_1 \parallel A_2:\)

\(\Sigma' = \emptyset\)

\(\text{foreach } \sigma \in \Sigma \text{ do}\)

\(\Sigma_1 := \text{check}(\{\sigma\}, A_1, \epsilon)\)

\(\Sigma_2 := \text{check}(\{\sigma\}, A_2, \epsilon)\)

\(\Sigma' \cup = (\Sigma_1 \oplus \Sigma_2)\)

\(\text{end}\)

\(\text{return } \text{check}(\Sigma', A_c, \epsilon)\)

end

Algorithm 1: Derive Initial Channel Sets

3.2 The Algorithm

The algorithm for generation of initial channel sets is depicted as Algorithm 1. Here \(X \cup = Y\) for sets \(X\) and \(Y\) means updating \(X\) to \(X \cup Y\).

Before introducing the algorithm, some data structure is needed. The state \(\sigma\) is refined to a map, which is an extension of the former one. For each role \(r, \sigma r\) is a pair of the form \((K, I)\), where \(K\) is the set of channels known by \(r\) under the state, \(I\) is the derived initial channel set of \(r\). We will use \((\sigma r').K\) and \((\sigma r').I\) to denote the two parts respectively. For the situation that an activity may be executed under different states because of non-determinism, we represent a set of states by \(\Sigma\).

Function “check” implements the algorithm. \(\text{check } (\Sigma, A, A_c)\) means handling activity \(A\) under states \(\Sigma\), and then the continuation \(A_c\) under the resulting states. If \(A\) is an interaction \(c_1 : r' \xrightarrow{c_2} r''\), the algorithm checks whether the sender knows \(c_1\) or \(c_2\). If the state shows that the sender does not know the
channel(s) at that time, we add the channel(s) to the initial channel set of $r^i$ (line 7 to 12). In any case, $c_2$ is added into the receiver’s known set (line 13).

The algorithm traces following the structure of the activity of a choreography. If a sequential activity $A_1; A_2$ is encountered, we handle $A_1$ firstly, and then $A_2$ under the resulting states (line 16). If a choice $A_1 \sqcap A_2$ is encountered, we handle each branch and the continuation separately, and finally return the superset of all results. If a parallel activity $A_1 \parallel A_2$ is encountered, for each possible state, we handle two branches respectively, and then merge the results of each branch by $\oplus$ operator (line 23 to 26). After merging, we handle the continuation after merging. $\Sigma_1 \oplus \Sigma_2$ means selecting any two states from $\Sigma_1$ and $\Sigma_2$ respectively, merging the known channel sets of each role in the two states, and so do the initial known channel sets.

$$\Sigma_1 \oplus \Sigma_2 \triangleq \{ \sigma \mid (\sigma r^i).K = (\sigma_1 r^i).K \cup (\sigma_2 r^i).K, (\sigma r^i).I = (\sigma_1 r^i).I \cup (\sigma_2 r^i).I, \sigma_1 \in \Sigma_1, \sigma_2 \in \Sigma_2 \}$$

To handle a choreography $[r^i[C_m^i], A_0]$, we set the initial state $\sigma_0 = \{ r^i \mapsto (\emptyset, \emptyset) \}$. Then we invoke check($\{\sigma_0\}, A_0, \epsilon$) and get final returned $\Sigma_0$. By merging all derived initial channel sets of a role, we can get all the channels which the role may be required to initialize, i.e., $\bigcup_{\sigma \in \Sigma_0} (\sigma r^i).I$ for role $r^i$.

### 3.3 Implementation

We have implemented the algorithm based on Pi4SOA[1], which is a open source tool for designing WS-CDL choreographies. We reuse the existing modules in Pi4SOA to parse a given choreography into a Java object, and then run our algorithm on the parsed choreography. Our program takes a “.cdl” or “.cdm” document as input, and prints the initial channel sets for each role. The program is available on our group home page\(^1\).

For a given choreography, the result produced by the algorithm can be used to aid generating role implementations, using the projection proposed in [7]. Also, all algorithms proposed in [15] can be applied to verify channel related properties such as channel-absence and channel redundancy. Furthermore, we will discuss how to use the algorithm to detect design defects such as logic mistakes in choreographies.

### 4 Checking Choreographies

Using the algorithm given in the last section, the implicitly assumed initial channel set for each role in a given choreography can be uncovered. By analyzing the results, we may get some clues that are helpful to detect design defects in choreographies. From our experience, the defects always show themselves in the form of unreasonable initial channel sets.

\(^1\) http://www.is.pku.edu.cn/~fmows/
We identify two kinds of suspicious initial channel sets, namely “cyclic dependency” and “duplicate declaration”. In these cases, the choreography worths examining more closely.

1. **Cyclic dependency.** Given a choreography, we say the initial channel sets of roles are cyclic dependant, if there are some roles form a cycle, each of them is required to know some channels belonging to the next role, e.g., role $r^1$ is required to know a channel of $r^2$, and $r^2$ is required to know a channel of $r^1$.

Cyclic dependency is impractical in most business processes. If a service knows a channel initially, it means the service will choose a partner and introduce it into a conversation. The situation may indicate some design defect in the original choreography. Here is an example. In an online shopping scenario, when a deal is agreed by a buyer $b$ and a seller $s$, the buyer sends channel $c_b$ to the seller, who forwards it to a shipper $d$. Then the shipper can contact with the buyer to arrange delivery details. However, suppose the designer forgets the channel passing and writes the following choreography.

$$Chor_1 = [ (b[[c_b]], s[[c_s]], d[[c_d]]), \quad I_1 : \quad c_s : b \rightarrow s; \quad I_2 : \quad c_d : s \rightarrow d; \quad I_3 : \quad c_b : d \rightarrow b ]$$

Our algorithm will produce the initial channel sets as follows.

$b : \{c_s\} \quad s : \{c_d\} \quad d : \{c_b\}$

It is surprising that the shipper knows a channel of the buyer initially, that means the shipper will always serve this buyer no matter what the seller says. This indicates that the choreography must have problems. In a well designed choreography, $I_1$ and $I_2$ may be “$c_s : b \rightarrow s$” and “$c_d : s \rightarrow d$” respectively.

2. **Duplicate declaration** means the situation where two roles are required to initialize a same channel. Sometimes, it is ridiculous if there is more than one role that initializes a same channel. Since different roles may be implemented and maintained by different organizations, it is probable that they assign different values to the channel and cause inconsistency. Let’s reconsider the example presented above, supposing the buyer can query the shipper for delivery schedule through interaction $I_4$.

$$Chor_2 = [ (b[[c_b]], s[[c_s]], d[[c_d]]), \quad I_1 : \quad c_s : b \rightarrow s; \quad I_2 : \quad c_d : s \rightarrow d; \quad I_3 : \quad (c_b : d \rightarrow b \parallel \quad I_4 : \quad c_d : b \rightarrow d) ]$$

The derived initial channel sets are:

$b : \{c_b, c_s, c_d\} \quad s : \{c_d\} \quad d : \{\}$

Please not that both the buyer and the seller can initialize the channel of the shipper. The implementations of them may have their own favorite shippers.
Suppose the buyer chooses FedEx and the seller asks DHL to deliver goods. Then, the buyer will always query FedEx for delivery time, who cannot answer the query because the seller let DHL take charge of the delivery.

4.1 Real Experience

We have applied our implementation to some example choreographies provided by the W3C CDL Work Group and Pi4SOA. The program worked well to calculate initial channel sets, and the results indeed helped us to find some design defects.

WS-CDL Primer [13] proposes a sample choreography which is used throughout the document. The same example in “cdm” format is included in Pi4SOA release package. The choreography describes an online shopping process. Using the syntax presented in Section 2, we rewrite the behavior of the choreography here, shown as Figure 2. There are four roles, BR (BuyerRoleType), SR (SellerRoleType), CR (CreditCheckerRoleType) and DR (ShipperRoleType). Four channels are involved, e.g. $B2SC$, which is short for $Seller2ShipperC$.

The process in the choreography has four steps. Firstly, a buyer requests a quote from a seller (line 1). In the second step, the buyer can respond the quote in three ways, either to give up the deal (line 2) by doing nothing but waiting for timeout, or to accept the quote and send a channel to the shipper for consequent delivery (line 3 to 6), or to bargain with the seller for another quote (line 7 to 8). It is originally a loop, which can be executed one or more times. We omit the loop, as if it is executed once, which does not affect the set of possible state. In the third step (line 10), the seller asks a credit checker to
validate the buyer’s credit. Finally, two different situation are handled. If the checker says no, the deal is over (line 11). If it says OK, the seller contacts a shipper and forwards channel DeliveryDetailsC (line 13 to 16). The shipper uses channel DeliveryDetailsC to inform the buyer about the details of the delivery (line 17).

The choreography seems quite reasonable, but our algorithm gives a negative result.

\[ BR : \{B2SC, DeliveryDetailsC\} \]
\[ SR : \{DeliveryDetailsC, S2CC, S2SC\} \]
\[ CR : \{\} \]
\[ DR : \{\} \]

The result shows that both the buyer and the seller are required to initialize “DeliveryDetailsC”, which is prepared by the buyer to receive messages from the shipper. There may have something wrong with the choreography, as we have said in the last section. After careful analysis, we found a design defect in the original choreography. In line 2, the buyer might choose “no bartering” to give up the deal. Then, the control flow jumps to line 10, where the seller continues to contact the credit checker and arrange delivery without recognizing that the buyer has gone. In standard business practice, the seller should give up the deal coordinated with the buyer. The choreography can be revised by moving the last two sections of codes to the end of line 5, which means the seller will only try to check customer’s credit after the buyer accepts the quote. The modified choreography seems more reasonable.

The example shows that it is indeed necessary to check channel passing in choreography design. Although the scenario only involves four roles and many complex mechanisms such as exception handling and finalization are not presented, we still find some mistake here.

5 Discussion

Our algorithm can uncover the underlying assumption of initial channel sets, which is helpful in validation, implementation and detection of some design defects in choreographies, it hard to say the problems are completely solved. We suggested two forms of suspicious assumptions, which are not sound, in the sense that not all choreographies with such kind of assumptions are incorrect. For example, in an auction scenario, both buyers and sellers should choose an auctioneer, and the buyer and sellers with the same choice may join in one conversation. So, it is the designers who can judge whether a role should know a channel initially, or whether the initial channel sets are reasonable. The judgment requires the underlying practice standard or application-specific knowledge.

Generally speaking, we propose an approach to derive initial channel sets of choreographies, and the results may reflect design defects and give some clues to find them. Application-specific knowledge is needed to confirm the problem and to identify the real defects. Thus, it is better to make these design decisions explicit.
We suggest some improvement for choreography languages such as WS-CDL. Initial channel set is orthogonal to most existing constructs, so it can be easily add into choreography language syntax. For the language presented in Section 2, it can be modified as below:

\[ RDecl ::= r[C_k, C_m] \]

Where \( r \) is the role name, \( C_k \) is a set of channels which the role must provide some channel instance to initialize, and \( C_m \) is a set of channels that the peer listens on.

With the construct, solutions presented in papers [15,7] can be applied to validation and implementing.

6 Conclusion and Future work

As a crucial mechanism to achieve dynamic Web services composition, channel passing is introduced in both choreography and orchestration languages. However, the choreography language WS-CDL is not ready for rigorous validation and implementation with respect to channel passing, since it lacks of a mechanism to specify the initial channel sets. In this paper, we develop a formal model of WS-CDL and present an algorithm to derive initial channel sets for roles. One theoretical application of this algorithm is to implement and verify choreographies together with channel passing. More importantly, we proposed an approach to detect design defects such as logic mistakes in choreographies under the assistance of the algorithm.

Besides theoretical results, we have implemented the algorithm in Java. The implementation can read WS-CDL documents in XML format, and then produce the initial channel sets for the roles in the choreographies. We applied our program and checking approach on some examples and indeed found some design defects, including the main example in WS-CDL Primer [13]. This shows the importance of validation with respect to channel passing and the effectiveness of our work.

We discussed a simple model of choreography where some features are excluded, for example exception handling, choreography performing, multiple participant instances, information driven control flow [3], service channels and session channels [9]. We plans to study channel passing in a full model including these future. Another possible future work is to explore the relationship between choreography and orchestration together with channel passing.

References