IJE TRANSACTIONS B: Applications Vol. 29, No. 11, (November 2016) 1549-1557

International Journal of Engineering

Journal Homepage: www.ije.ir

Web Service Choreography Verification Using Z Formal Specification

Y. Rastegari*, Z. Sajadi, F. Shams

Department of Computer Science and Engineering, Shahid Beheshti University, Tehran, Iran

PAPER INFO

Paper history:
Received 30 November 2015
Received in revised form 12 September 2016
Accepted 30 September 2016

Keywords:
Web Service Choreography
Compatibility
Verification
Adaptation
Z Formal Specification

ABSTRACT

Web Service Choreography Description Language (WS-CDL) describes and orchestrates the services interactions among multiple participants. WS-CDL verification is essential since the interactions would lead to mismatches. Existing works verify the messages ordering, the flow of messages, and the expected results from collaborations. In this paper, we present a Z specification of WS-CDL. Besides verifying the mentioned concerns, we find out whether the choreographies are realizable by web services protocols at orchestration level. In this regard we detect the interactions between each two distinct participants which lead to deadlock or unspecified reception. An 'itinerary purchase' case study for prototyping the transformation rules is presented and the Z/EVES tool is used to demonstrate the protocol compatibility. Also, we define multiple attributes to compare the choreography description languages/models from the verification and adaptation viewpoints.

doi: *10.5829/idosi.ije.2016.29.11b.08*

1. INTRODUCTION

Choreography describes peer-to-peer collaborations of service consumers and service providers (i.e. choreography participants) from a global viewpoint. Choreography defines ordered message exchanges which result in accomplishing a common business goal. Web Service Choreography Description Language (WS-CDL) [1] is the W3C recommended language for describing service choreographies.

WS-CDL is protocol-compatible if every joint execution of each two distinct participants leads to a proper final state, i.e. a state in which both participants are in a final state in their respective protocols. Protocol mismatches are defined in two main types: unspecified reception and deadlock [2]. Unspecified reception occurs when one party sends a message while the other is not expecting it. Deadlock refers to the case where both parties are mutually waiting to receive some messages from the other. WS-CDL has a static structure and does not consist of dynamic elements and management rules which govern the behaviors of participants; therefore it is essential to transform WS-CDL into adaptable and verifiable models.

There are two types of transformation including model-driven [3-6] (with the goal of adaptation) and formal [7-11] (with the goal of verification) in the literature. The model driven approaches translate a WS-CDL element to its respective replacement in terms of BPEL as well as WSCDL. This enables tracing down changes from choreography to orchestration and vice versa which is an important issue in the choreography adaptation scope. On the other hand, some studies formalize the WS-CDL elements. They tried to verify several aspects of service choreography like protocol compatibility, time constraints, and message ordering. It might also be observed that these works are limited to a specific subject and does not check whether the committed choreography is realizable by the existing services protocols at the orchestration level.

In this paper, we aim at transforming WS-CDL into Z models that are modifiable to overcome new requirements and also verifiable to prevent unexpected faulty behaviors that are mentioned in the related studies. Furthermore, we detect the interactions between each two distinct participants which lead to deadlock or unspecified reception. This is significant because web services protocols defined in WS-BPEL processes,

*Corresponding Author's Email: y_rastegari@sbu.ac.ir (Y. Rastegari)

Please cite this article as: Y. Rastegari, Z. Sajadi, F. Shams, Web Service Choreography Verification Using Z Formal Specification, International Journal of Engineering (IJE), TRANSACTIONS B: Applications Vol. 29, No. 11, (November 2016) 1549-1557

underlie and realize the dependent WS-CDL specification.

The rest of this paper is organized as follows. We explain the related studies and compare those regarding adaptation and verification issues in section 2. Section 3 describes an overview of WS-CDL specification and presents the itinerary purchase scenario as a running example. We present formal specification of WS-CDL and discuss about the rationale behind the transformation of each element in section 4. Section 5 verifies the correctness of the transformation and the compatibility of services protocols. The paper is concluded in section 6.

2. RELATED WORK

There are two types of transformation including model-driven [3-6] (with the goal of adaptation) and formal [7-11] (with the goal of verification) in the literature. Verification is used to check the process (application) consistency, after performing adaptation actions. If we consider MAPE (Monitor, Analysis, Plan, Execute) feedback loop, verification is performed after planning for suitable adaptation actions and before executing the actions. Verification checks whether the adaptation actions preserve process (application) consistency or not.

The model driven approaches translate a WS-CDL element to its respective replacement in terms of BPEL as well as WSCDL. This enables tracing down changes from choreography to orchestration and vice versa which is an important issue in the choreography adaptation scope.

The formal approaches verify several aspects of service choreography like protocol compatibility, time constraints, and message ordering.

Mendling el al. [3] proposed a model driven transformation approach to drive BPEL process definitions from a global WS-CDL model. The approach includes a mapping between WS-CDL and WS-BPEL building blocks. In addition, the mapping can be used to generate WS-CDL description from existing WS-BPEL processes. In another model-driven approach, CDL2BPEL [4] algorithm translates WS-CDL to "BPEL and WSDL" elements, according to a knowledge base. The knowledge base contains generic patterns to translate a WS-CDL entity to its respective replacements in terms of BPEL as well as optional WSDL. The algorithm extracts WSDL interfaces from interactions and "tokens / token locators". BPEL4Chor [5] is an intermediary language to align choreography and orchestration. BPEL4Chor is a non-executable choreography language, forming an additional layer on top of the BPEL standard [6].

The transforming of source models to formal specifications is addressed in some works with the goal of quality evaluation [12, 13]. Nematzadeh and Nematzadeh [7] proposed the mapping rules from eflow and BPEL to colored petri net for reliability and performance measurement. In reference [8], a simple CDL is introduced to formalize the WS-CDL's participant roles, and the collaborations among roles. They used SPIN model-checker to reason about properties that should be satisfied by the specified system automatically. Furthermore, in order to verify WS-CDL protocol mismatches, the transformation rules were proposed to correspond the WS-CDL entities with timed automata [9], and colored petri-net [10, 11] elements. These formal languages are suitable for choreography verification, but they cannot realize the requirements of an adaptive process. For example, CDL and timed automata do not support all workflow patterns; colored petri-net does not support the separation of business logic and implementation code, nor abstract modeling, nor distinct control model.

From the adaptation and verification viewpoints, we consider the below attributes to compare the choreography description languages/models. The comparison results are shown in Table 1.

1) Structure

- Dynamism: Dynamic structure means that the structure of a process must be flexible to being reconfigured and regulated dynamically in response to the management rules.
- Workflow support: It refers to both supporting of workflow and services interaction patterns (e.g. sequence, parallel, synchronization, sending, receiving, etc.).
- Hierarchical (nested): A hierarchical process is designed level-by-level in order to hide the unnecessary details at each abstraction level. At each level, there is a composite operation that may be broken down at the next lower level.
- Separation of concerns: Separation of concerns enables the separate development of the business logic, and the crosscutting concerns of a process (e.g. quality of service, implementation code) [14].

2) Control

- Reconfigurable: It refers to modifying the structure and runtime behaviors of a process by management rules.
- Verifiable: Choreography verification consists of two main types of protocol mismatches. Service interoperability verification which includes message ordering and time constraints at design time [11] and deadlock, in which both parties are mutually waiting to receive some messages from the other [15].

3. AN OVERVIEW OF WS-CDL

As shown in Figure 2, a choreography element contains activity, exception handling and finalizer parts.

TABLE 1. The comparison of choreography modeling and description languages

Language / Model	Goal	Structure				Control	
		Dynamism	Workflow support	Hierarchical	Separation of concerns	Reconfigurable	Verifiable
WS-CDL [1]	Specification	-	●	✔	-	-	-
WSCI [16]	Specification	-	●	-	-	-	-
BPEL4Chor [8]	Specification	-	●	✔	-	-	-
CDL [13]	Specification Verification	-	-	-	-	●	✔
Timed automata [14]	Verification	✔	●	-	-	●	●
Colored Petri net [15,16]	Verification	✔	✔	●	●	●	●
Z (our work)	Specification Verification	✔	✔	✔	●	●	✔

✔ Complete support | ● Partial support | - Lack of support

Choreography: The attribute name specifies a distinct name for a choreography element. The root choreography is the only choreography that is enabled by default; it performs other non-root choreographies subsequently.

Activity: Activities describe the actions performed within a choreography. The activity notation is used to define basic actions, ordering structures, and work-unit of activities. The activity notation provides all required elements for describing services interactions, ordering of interactions, and choreography composition.

Exception handling: The exception block is used to handle performance failures. The failures emerge while an exceptional circumstance or an error occurs, like interaction or security failures, timeout or validation errors, etc. When an exception occurs, a work-unit within the exception block is performed.

Finalizer: The finalizer block is enabled when a choreography is successfully completed. The activities within a finalizer block are performed to confirm, cancel or modify the effects of completed actions.

Here we adopt the 'itinerary purchase' collaborative business process [17] for prototyping the transformation rules. The itinerary purchase process is handled by the following independent and collaborating parties: Customer, Travel agency, Airline, Hotel, and Payment system.

Figure 2 shows the 'itinerary purchase' process model in BPMN choreography notation. The itinerary purchase scenario is as follows. (1) First, the customer requests the travel agency for available itineraries, and then the travel agency sends all available itineraries to the customer. (2) Next, the customer selects the desired itinerary and requests the travel agency for reservation. (3) The travel agency starts two parallel choreographies with the hotel and the airline parties, and waits until reservation responses arrive. If both of the reservations are done, then the travel agency calculates the total cost of itinerary. (4) After the total cost is determined, the choreography between the travel agency and the payment system is started. Again, the travel agency waits until the payment is confirmed by the payment system. (5) Finally, a choreography is started to notify the customer about the purchase status. Figure 1 shows the specification of 'itinerary purchase' process in WS-CDL format.

4. TRANSFORMATION

4. 1. Ordering Structures Ordering structures are used to combine activities and express the ordering rules of actions. WS-CDL presents the Sequence, Parallel and Choice ordering structures. An ordering structure can include other ordering structures recursively; hence an activity is combined with other ordering structures in a nested way.

Sequence: The activities within a sequence element must be performed one after another. After transforming enclosing activities to their corresponding operation schemas, the composition operator could be used to perform the operations sequentially. The sequence element in the 'itineraryPurchase' choreography is transformed to the following specification:

$sequenceOp1 \triangleq getItineraries \fatsemi requestReservation \fatsemi itineraryReservation \fatsemi paymentProcessing$

Parallel: The activities within a parallel element are enabled concurrently. The parallel activity completes successfully when all its enclosed activities complete successfully.

```
<choreography name="itineraryPurchase" root="true">
  <sequence>
/* (1) customer , travel agency */
    <interaction name="itinerary" operation="getItineraries">
      <participate relationshipType="Customer_TravelAgency"
       fromRole="CustomerRole" toRole="TravelAgencyRole" />
      <exchange name="requestItineraries" action="request">
        <send variable="tripProfile"/>
        <receive variable="tripProfile"/>
      </exchange>
      <exchange name="itinerariesList" action="respond">
        <send variable="itinerariesList"/>
        <receive variable="itinerariesList"/>
      </exchange>
    </interaction>

/* (2) customer , travel agency */
    <perform choreographyName="requestReservation"></perform>
/* (3) travel agency , airline | travel agency , hotel */
    <perform choreographyName="itineraryReservation"></perform>
/* (4) travel agency , payment system */
    <perform choreographyName="paymentProcessing"></perform>
  </sequence>

  <exceptionBlock name="exceptionHandling">
    <workunit guard="cancel">
      <sequence>
        <perform choreographyName="itineraryCancelation"></perform>
        <perform choreographyName="cancelNotification"></perform>
      </sequence>
    </workunit>

    <workunit guard="handleTimeout">
      <noAction>
    </workunit>
  </exceptionBlock>

  <finalizerBlock>
    <workunit name="finalizing">
/* (5) travel agency , customer */
      <perform choreographyName="successNotification"></perform>
    </workunit>
  </finalizerBlock>
</choreography>

<choreography name="itineraryReservation">
  <parallel>
/* (3.1) travel agency , airline */
    <perform choreographyName="flightReservation"></perform>
/* (3.2) travel agency , hotel */
    <perform choreographyName="roomReservation"></perform>
  </parallel>
</choreography>
```

Interaction block

Exception block

Finalizer block

Figure 1. The specification of "itinerary purchase" process in WS-CDL format

After transforming enclosing activities to their corresponding operation schemas, the conjunction operator could be used to perform the operations in parallel. The parallel element in 'itineraryReservation' choreography is transformed to the following specification:

expandedItineraryReservation ≙ *flightReservation* ∧ *roomReservation*

Choice: The choice ordering structure realizes a dynamic conditional branch. Although the choice element encompasses one or more activities, only one activity is selected and the other activities are disabled. After transforming enclosing activities to their corresponding operation schemas, the exclusive-or operator could be used to perform only one operation at a time.

4. 2. Basic Activities A basic activity provides the lowest level actions for service interaction, choreography composition, and describing silent/hidden activities. It also provides building blocks for handling exceptions, and finalizing choreographies.

Interaction: Interaction is the most important activity of the WS-CDL specification. It leads to an information exchange between participants. In fact, an interaction is a pair of message exchanges for delivering data between a consumer and a provider, and defining the actual values of the delivered data. Furthermore, an interaction specifies the service operation that should be consumed to prepare the response message. An interaction is initiated when the consumer sends a message to the provider. Meanwhile, the provider performs the requested operation, and

responds with a normal response message or a fault message.

To describe the interaction in Z, first we describe system state schema and initialization schema. Then each exchange element is transformed to Z operation schema which will be performed sequentially by composition operator. The 'Action' free type defines the right action type of each exchange. The 'RoleType' free type defines the collaborating parties. The 'allMessages' free type defines all messages of collaborations. We used 'OrderedMessages' axiom to show the valid order of messages. In 'itineraryPurchase' state schema, 'customer' and 'travelAgency' are two sequences of ordered pair of action type and message. To describe the relation between services, the channel state variable is used in the declaration part of 'itineraryPurchase' schema. The 'msg' state variable is used to show the right message exchange between right participants. The state variable 'exchange_message' is a subset of 'OrderedMessages' and represents the current message with its order number. The order number is used to check the message ordering.

Action ::= *send* | *receive*

RoleType ::= *customerRole* | *travelAgencyRole*

allMessages ::= *init* | *tripProfile* | *itinerariesList* | *selectedItinerary* | *selectedAirline* | *selectedHotel* | *airlineConfirm* | *hotelConfirm* | *paymentProfile* | *paymentConfirm* | *notifySuccess* | *notifyCancel*

OrderedMessages == {*0*↦*init*, *1*↦*tripProfile*, *2*↦*itinerariesList*, *3*↦*selectedItinerary*, *4*↦*selectedAirline*, *4*↦*selectedHotel*, *5*↦*airlineConfirm*, *5*↦*hotelConfirm*, *6*↦*paymentProfile*, *7*↦*paymentConfirm*, *8*↦*notifySuccess*, *8*↦*notifyCancel*}

To describe the interaction, first we describe the 'itineraryPurchase' state schema and the 'itineraryPurchaseInit' initialization schema.

_____*ItineraryPurchase*_____

customer:seq(*Action* × *allMessages*)

travelAgency:seq(*Action* × *allMessages*)

channel:*RoleType*↔*RoleType*

msg:*allMessages*→(*RoleType*↔*RoleType*)

ole}}

travelAgency′=travelAgency⁀⟨(*send*,*itinerariesList*)⟩

customer′=customer⁀⟨(*receive*,*itinerariesList*)⟩

The 'requestItineraries' operation schema and the 'itinerariesList' operation schema are performed sequentially by the composition operator.

getItineraries ≙ *requestItineraries*⨾ *itinerariesList*

No-action, Silent-action: The no-action and the silent-action activities are used when a participant does not perform any action, or perform an action without any observable operational details, respectively. Since no-action does not change the 'itineraryPurchase' state, the following operation schema describes its logic in Z:

_____*noAction*_____

exchange_message:*OrderedMessages*

act:*allMessages*

last_msg:*allMessages*

dom(*msg*)=ran(*exchange_message*)

msg={*act*↦*channel*}

last_msg={*notifySuccess*} ∨ *last_msg*={*notifyCancel*}

#(last_msg)=1

To initialize the system state variables, the 'itineraryPurchaseInit' is described as follows:

_____*itineraryPurchaseInit*_____

itineraryPurchase′

customer′=⟨⟩

travelAgency′=⟨⟩

channel′=∅

exchange_message′={0↦*init*}

msg′=∅

act′=*init*

We describe the 'requestItineraries' exchange block with the action type of 'request', by the following operation schema:

_____*requestItineraries*_____

Δ*itineraryPurchase*

dom(*exchange_message*)={0}

channel′={*customerRole*↦*travelAgencyRole*}

act′=*tripProfile*

exchange_message′={1↦*tripProfile*}

msg′={*tripProfile*↦{*customerRole*↦*travelAgencyRole*}}

customer′=customer⁀⟨(*send*,*tripProfile*)⟩

travelAgency′=travelAgency⁀⟨(*receieve*,*tripProfile*)⟩

Similarly, we describe the 'itinerariesList' exchange block with the action type of 'respond', by the following operation schema:

_____*itinerariesList*_____

Δ*itineraryPurchase*

dom(*exchnage_message*)={1}

channel′={*customerRole*↦*travelAgencyRole*}

act′=*itinerariesList*

exchange_message′={2↦*itinerariesList*}

msg′={*itinerariesList*↦{*customerRole*↦*travelAgencyR*

Ξ*itineraryPurchase*

Perform: The perform activity enables a choreography to reuse and combine other existing choreographies hierarchically. It has 'name' attribute for referencing the name of the choreography to be performed. In our example, the 'itineraryReservation' perform element, is transformed to the following schema:

_____*itineraryReservation*_____

expandedItineraryReservation

The 'expandedItineraryPurchase' performs the 'flightReservation' choreography and the 'roomReservation' choreography in parallel.

expandedItineraryReservation ≙ *flightReservation* ∧ *roomReservation*

Exception block, Finalizer block: The exception handling block and the finalizer block are described in section 3. The exception block contains one or more work-units, each work-unit handles an exceptional circumstance. The finalizer block contains required activities for finalizing its enclosing choreography performance. After transforming enclosing work-units to their corresponding operation schemas, the exclusive-or operator could be used to perform only one work-unit at a time. In our example, the 'exceptionBlock' encloses two work-units to handle the cancel notification and the timeout error. So, we defined the following specification:

$exceptionHandling \triangleq$
$(exceptionHandlingCancel \vee$
$exceptionHandlingTimeout)$
$\wedge$
$\neg(exceptionHandlingCancel \wedge$
$exceptionHandlingTimeout)$

4. 3. Work-unit A work-unit encloses activities, and defines the constraints that should be fulfilled to perform them. A work-unit has the 'guard' attribute for specifying the condition of variables in XPATH format. If the guard condition of a work-unit is satisfied, then its enclosed activities are enabled. In our example, the 'success' work-unit, the 'cancel' work-unit and the 'handleTimeout' work-unit are transformed to the following operation schemas:

_____finalizer_____
$\Xi itineraryPurchase$
guard?:*String*
guard?=*success*⇒*successNotification*
_____exceptionHandlingCancel_____
$\Xi itineraryPurchase$
guard?:*String*
guard?=*cancel*⇒*itineraryCancelation*
cencelNotification
_____exceptionHandlingTimeout_____
$\Xi itineraryPurchase$
guard?:*string*
guard?=*handleTimeout*⇒*noAction*

4. 4. Total Specification After transforming each WS-CDL element to its respective Z element, the 'Itinerary Purchase' process is defined by the following formal specification:

$T\_itineraryPurchase \triangleq (sequencOp1 \wedge$
$exceptionHandling) \vee$
$(sequenceOp1 \wedge finalizer)$

If an exception occurs while performing the 'sequenceOp1', then the 'exceptionHandlingCancel' or the 'exceptionHandlingTimeout' is enabled, otherwise, the 'finalizer' operation schema is performed to finalize the process performance.

5. VERIFICATION

5. 1. Correctness In this section, we describe semantic preservation of Z models to prove the total correctness of proposed transformation rules. The semantic of source models (i.e. WS-CDL) is preserved, if transformation rules produce behaviorally equivalent target models (i.e. Z). In the following list, we show that our proposed transformation rules preserve the message ordering, the flow of messages, and the expected results.

- WS-CDL's ordering and composing structures are corresponded with Z elements in a straightforward form (see the transformation rules in section 4).
- To control the flow of messages, WS-CDL uses guard conditions in exception block and work-unit. Similarly, Z controls the flow of messages by evaluating guards associated with schemas.
- The 'exchange_message' variable preserves the message ordering as defined in WS-CDL. We define the following Z specification to preserve the order of messages in our example:

$OrderedMessages == \{0 \mapsto init, 1 \mapsto tripProfile,$
$2 \mapsto itinerariesList, 3 \mapsto selectedItinerary,$
$4 \mapsto selectedAirline, 4 \mapsto selectedHotel,$
$5 \mapsto airlineConfirm, 5 \mapsto hotelConfirm,$
$6 \mapsto paymentProfile, 7 \mapsto paymentConfirm,$
$8 \mapsto notifySuccess, 8 \mapsto notifyCancel\}$
$exchange\_message \in OrderedMessages$

- It is necessary to prove whether the messages are exchanged between the right source and destination web services. To prove this property, we define two following Z axioms in our example:

$dom(msg) = ran(exchange\_message)$
$msg = \{act \mapsto channel\}$

- The last message of choreography represents the expected results. To verify the last message of itinerary purchase process, we define the following axioms:

$last\_msg \in allMessages$
$last\_msg = \{notifySuccess\} \vee last\_msg = \{notifyCancel\}$
$\#(last\_msg) = 1$

5. 2. Protocol Compatibility After describing choreography commitment in Z, it is necessary to check whether the participants could realize the commitment regarding their local processes and the order of messages they send and receive.

If we consider a multi-party choreography and restrict it to those interactions that involve a given pair of service - e.g. the interactions between the customer and the travel agency in our example - we obtain a bilateral service protocol. Two services are protocol-compatible if every joint execution of these services leads to a proper final state, i.e. a state in which both services are in a final state in their respective protocols

[18]. Yellin & Strom [2] identified two main types of protocol mismatches: unspecified reception and deadlock. Unspecified reception occurs when one party sends a message while the other is not expecting it. Deadlock refers to the case where both parties are mutually waiting to receive some message from the other. Figure 3 illustrates the protocol mismatches and their detection patterns (consider the protocols Ps of service Ss, and Pc of service Sc). As shown in Figure 3(a), Ps expects to receive message c after sending a, while Pc is waiting to receive b; this is a deadlock case. On the other hand in Figure 3(b), Ps sends message b while Pc does not expect to receive it; this is an unspecified reception case.

To detect the mentioned protocol mismatches we applied the detection patterns proposed by Motahari Nezhad, et al. [18]. They decomposed protocol tree into distinct paths. Then the best candidate pair of messages is considered as a reference pair (RP) in the same path-pair. For example, Figure 3(a) shows two paths from protocol Ps and Pc. The message pair -c and +c are selected as a reference pair. We use reference pair to check the order of exchanging messages and find out the mismatches as described in the following patterns.

Deadlock detection pattern: As shown in Figure 3(a), given reference pair +c and -c the candidate matching pair -b and +b is called a conflicting match. This is because -b (an outgoing message) with a bigger depth than +b (an incoming message) leads to a deadlock in the interaction of two services in case this matching is allowed.

Unspecified reception detection pattern: As shown in Figure 3(b), given reference pair -b and +b the candidate matching pair -a and +a is called a conflicting match. This is because -a (an outgoing message) with a bigger depth than +a (an incoming message) leads to an unspecified reception in the interaction of two services in case this matching is allowed.

To describe the above patterns in Z, we define two sequences of ordered pair Ps and Pc in which their domain define the operation type (send or receive) and their range define the exchanging messages between two web services (e.g. a, b, c, etc.).

To detect the deadlock mismatch, we search for two pairs which have the same range and unequal domains. They are called reference pairs (e.g. +c and -c). Then we search for pairs which have the same range and unequal domains, from the RP to the end of sequence Ps, and from the beginning of Pc to the RP (e.g. –b and +b). We call these pairs conflicting pairs. The domain of CP in Ps, and the domain of CP in Pc must be unequal with the domain of RP in the relevant path. Also the domain of RP in Pc must be from send (-) type.

Detecting the unspecified reception mismatch is the same as deadlock, where it is expected that the domain of CP in Ps, and the domain of CP in Pc must be equal with the domain of RP in the relevant path. Also the domain of RP in Pc must be from receive (+) type. The formal specification of the deadlock detection pattern and the unspecified reception detection pattern are shown below.

Deadlock

$$\exists\ i, j, x, y:\mathbb{Z} \mid i\in 1..\#Ps \land j\in 1..\#Pc \land x\in 1..j\text{-}1 \land y\in i\text{+}1..\#Ps \bullet$$
$$\mathrm{dom}\{(Ps_i)\}\neq\mathrm{dom}\{(Pc_j)\} \land \mathrm{ran}\{(Ps_i)\}=\mathrm{ran}\{(Pc_j)\} \land$$
$$\mathrm{dom}\{(Ps_y)\}\neq\mathrm{dom}\{(Pc_x)\} \land \mathrm{ran}\{(Ps_y)\}=\mathrm{ran}\{(Pc_x)\} \land$$
$$\mathrm{dom}\{(Ps_i)\}\neq\mathrm{dom}\{(Ps_y)\} \land \mathrm{dom}\{(Pc_j)\}\neq\mathrm{dom}\{(Pc_x)\} \land$$
$$\mathrm{dom}\{(Pc_j)\}=\{send\}$$

Unspecified reception

$$\exists\ i, j, x, y:\mathbb{Z} \mid i\in 1..\#Ps \land j\in 1..\#Pc \land x\in 1..j\text{-}1 \land y\in i\text{+}1..\#Ps \bullet$$
$$\mathrm{dom}\{(Ps_i)\}\neq\mathrm{dom}\{(Pc_j)\} \land \mathrm{ran}\{(Ps_i)\}=\mathrm{ran}\{(Pc_j)\} \land$$
$$\mathrm{dom}\{(Ps_y)\}\neq\mathrm{dom}\{(Pc_x)\} \land \mathrm{ran}\{(Ps_y)\}=\mathrm{ran}\{(Pc_x)\} \land$$
$$\mathrm{dom}\{(Ps_i)\}=\mathrm{dom}\{(Ps_y)\} \land \mathrm{dom}\{(Pc_j)\}=\mathrm{dom}\{(Pc_x)\} \land$$
$$\mathrm{dom}\{(Pc_j)\}=\{receive\}$$

The deadlock scenario and the unspecified reception scenario are described in the following state schemas, according to Figure 3. We use invariant theorems in Z/EVES tool [19] to demonstrate how to verify these protocol mismatch scenarios.

First we describe the order of messages between Ps and Pc, which leads to a deadlock case. Consider Ps as customer, and Pc as travel agency in the itinerary purchase process.

deadlockScenario

$\Delta stateSchema$

$channel' = \{Ps \mapsto Pc\}$

$Ps' = Ps^\frown (send,a)^\frown (receive,c)^\frown (send,b)$

$Pc' = Pc^\frown (receive,b)^\frown (receive,a)^\frown (send,c)$

We also describe the order of messages between Ps and Pc, which leads to an unspecified reception case.

unspecifiedReceptionScenario

$\Delta stateSchema$

$channel' = \{Ps \mapsto Pc\}$

$Ps' = Ps^\frown (send,b)^\frown (send,a)$

$Pc' = Pc^\frown (receive,a)^\frown (receive,b)$

Figure 4 shows the proof of deadlock, and Figure 5 shows the proof of unspecified reception. Z/EVES verifies the syntax of specifications and proves the theorems. The ‘Y’ character in the Proof column meaning that the scenario leads to the corresponding mismatch.

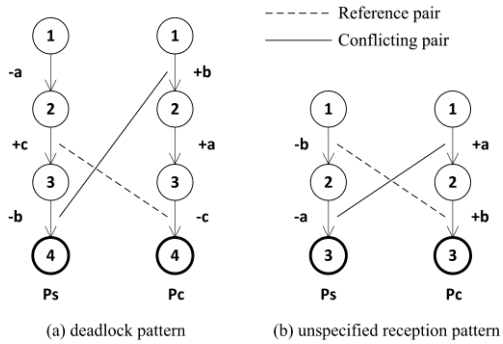

Figure 3. Protocol mismatches and their detection patterns

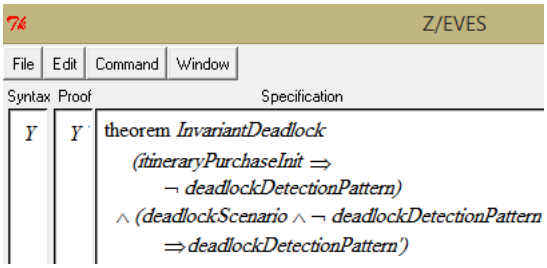

Figure 4. Invariant deadlock theorem

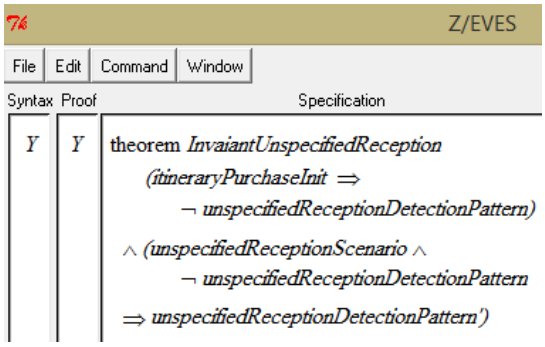

Figure 5. Invariant unspecified reception theorem

6. CONCLUSION

In this paper, we described the WS-CDL standard using the Z formal language. We presented the transformation rules and the rationale behind each rule. The benefits of this transformation include:

- Z is useful for both specification and verification of collaborative business processes.
- Z supports process hierarchy in which a process activity could be expanded in the lower levels. Therefore it is possible to transform both WS-CDL and WS-BPEL into their corresponding Z specifications, and integrate them in a nested way.
- Since Z is a verifiable language, the process designer could verify the processes and prevent them from mismatches during performance.

When a new requirement arises at choreography-level, it must be realized at orchestration-level. Therefore, the adaptive model must cover all choreography, and orchestration entities in different abstraction levels, and also consider the interoperability between them. In future, we will present a Z specification for BPEL language and create the interoperability between choreography and orchestration entities. This could be done with the help of hierarchical attribute of Z language. Also, we will try to deploy the state schemas at the Meta level, and their corresponding source code at the base level according to the reflective-state design pattern [20]. We consider concrete states and concrete services to realize the functionalities that are defined at the Meta level. Consequently, the adaptation designer (or an automatic adaptation unit) could easily modify the Meta level's state schemas, which mirror the system functionalities.

7. REFERENCES

1. Kavantzas, N., "Web services choreography description language (ws-cdf) version 1.0", ***http://www. w3. org/TR/ws-cdl-10/***, (2004).
2. Yellin, D.M. and Strom, R.E., "Protocol specifications and component adaptors", ***ACM Transactions on Programming Languages and Systems (TOPLAS)***, Vol. 19, No. 2, (1997), 292-333.
3. Mendling, J. and Hafner, M., "From ws-cdl choreography to bpel process orchestration", ***Journal of Enterprise Information Management***, Vol. 21, No. 5, (2008), 525-542.
4. Weber, I., Haller, J. and Mulle, J.A., "Automated derivation of executable business processes from choreographies in virtual organisations", ***International Journal of Business Process Integration and Management***, Vol. 3, No. 2, (2008), 85-95.
5. Decker, G., Kopp, O., Leymann, F. and Weske, M., "Bpel4chor: Extending bpel for modeling choreographies", International Conference on Web Services, IEEE., (2007), 296-303.
6. Weib, A., Karastoyanova, D., Molnar, D. and Schmauder, S., "Coupling of existing simulations using bottom-up modeling of choreographies", GI-Jahrestagung., (2014), 101-112.
7. Nematzadeh, H. and Nematzadeh, Z., "Deterministic measurement of reliability and performance using explicit colored petri net in business process execution language and eflow", ***International Journal of Engineering-Transactions A: Basics***, Vol. 28, No. 10, (2015), 1439.
8. Hongli, Y., Xiangpeng, Z., Zongyan, Q., Geguang, P. and Shuling, W., "A formal model for web service choreography description language (WS-CDL)", ***School of Mathematical Science. Peking University***, (2006).
9. Diaz, G., Pardo, J.-J., Cambronero, M.-E., Valero, V. and Cuartero, F., "Automatic translation of WS-CDL choreographies to timed automata, in Formal techniques for computer systems and business processes." (2005), 230-242.
10. Valero, V., Macia, H., Pardo, J.J., Cambronero, M.E. and Díaz, G., "Transforming web services choreographies with priorities and time constraints into prioritized-time colored petri nets", ***Science of Computer Programming***, Vol. 77, No. 3, (2012), 290-313.
11. Benabdelhafid, M.S. and Boufaida, M., "Toward a better interoperability of enterprise information systems: A cpns and

timed cpns-based web service interoperability verification in a choreography", ***Procedia Technology***, Vol. 16, (2014), 269-278.

12. Motameni, H. and Nemati, M., "Mapping crc card into stochastic petri net for analyzing and evaluating quality parameter of security", ***International Journal of Engineering-Transactions B: Applications***, Vol. 27, No. 5, (2013), 689-696.
13. Mhamdi, L., Dhouib, H., NSimen, A. and Liouane, N., "Using interval petri nets and timed automata for diagnosis of discrete event systems (des)", ***International Journal of Engineering-Transactions A: Basics***, Vol. 27, No. 1, (2014), 113-122.
14. McKinley, P.K., Sadjadi, S.M., Kasten, E.P. and Cheng, B.H., "A taxonomy of compositional adaptation", ***Rapport Technique numeroMSU-CSE-04-17***, (2004).
15. Kongdenfha, W., Motahari-Nezhad, H.R., Benatallah, B. and Saint-Paul, R., "Web service adaptation: Mismatch patterns and semi-automated approach to mismatch identification and adapter development", Web services foundations. (2014), 245-272.
16. Arkin, A., Askary, S., Fordin, S., Jekeli, W., Kawaguchi, K., Orchard, D., Pogliani, S., Riemer, K., Struble, S., Takacsi-Nagy, P. and Stand. Propos. by BEA Syst. Intalio, S., Sun Microsystems, . "Web service choreography interface (WSCI) 1.0", (2002).
17. Douglas, A., " "Ws-cdl eclipse," (2013). [online]. Available: Http://sourceforge.Net/projects/wscdl-eclipse/".
18. Motahari Nezhad, H.R., Xu, G.Y. and Benatallah, B., "Protocol-aware matching of web service interfaces for adapter development", in Proceedings of the 19th international conference on World wide web., (2010), 731-740.
19. Saaltink, M. and Canada, O., "The z/eves 2.0 user's guide", (1999).
20. Ferreira, L.L. and Rubira, C.M., "The reflective state pattern", ***Proceedings of the Pattern Languages of Program Design, TR# WUCS-98-25, Monticello, Illinois-USA***, (1998).

Web Service Choreography Verification Using Z Formal Specification

Y. Rastegari, Z. Sajadi, F. Shams

Department of Computer Science and Engineering, Shahid Beheshti University, Tehran, Iran

P A P E R I N F O

Paper history:
Received 30 November 2015
Received in revised form 12 September 2016
Accepted 30 September 2016

Keywords:
Web Service Choreography
Compatibility
Verification
Adaptation
Z Formal Specification

چکیده

زبان توصیف هم‌آرایی وب‌سرویس‌ها (WS-CDL) به منظور توصیف تعاملات و هماهنگی بین چندین واحد همکار استفاده می شود. ناسازگاری‌هایی ممکن است در زمان تعامل رخ دهند، بنابراین ضرورت دارد تا سند هم‌آرایی سرویس‌ها در زمان طراحی و یا بعد از تغییر و قبل از اجرای مجدد، درستی‌سنجی شود. کارهای مرتبط درستی‌سنجی غیرجامعی برای ترتیب پیام‌ها، جریان منطقی پیام‌ها و نتایج موردانتظار انجام داده‌اند. در تحقیق جاری، توصیف صوری سند هم‌آرایی با استفاده از زبان Z ارائه شده است. علاوه‌بر ارائه روشی جامع برای پوشش روش‌های پیشین، ما تحقق‌پذیری سند هم‌آرایی توسط پروتکل تعاملی سرویس‌ها را نیز درستی‌سنجی می‌کنیم. در واقع بررسی می‌شود که آیا پروتکل تعاملی موجود بین هر دو واحد همکار در سطح هم‌نوایی، منجر به توافقات صورت پذیرفته در سطح هم‌آرایی می‌شود یا خیر. در این راستا با توصیف الگوهای تعاملی توانستیم تعاملاتی که منجر به بن‌بست و یا پذیرش نامشخص می‌شوند را شناسایی کنیم. نحوه تبدیل سند هم‌آرایی به توصیف‌های صوری با استفاده از مطالعه موردی "فرآیند خرید برنامه سفر" ارائه گردید و از ابزار Z/EVES برای نمایش نحوه شناسایی ناسازگاری‌ها استفاده شد. در ضمن با تعریف معیارهایی به مقایسه مدل‌ها و زبان‌های توصیف هم‌آرایی از نقطه نظر تطبیق‌پذیری و درستی‌سنجی پرداختیم.

***doi**: 10.5829/idosi.ije.2016.29.11b.08*