Running head: TRANSFORMATION OF SERVICE CHOREOGRAPHY INTO UML-SM

Toward automatic Transformation of Service Choreography into UML State Machine

Yousef Rastegari[1], Fereidoon Shams[2]

[1] Shahid Beheshti University, Electrical & Computer Engineering Department, Tehran, Iran

[2] Shahid Beheshti University, Electrical & Computer Engineering Department, Tehran, Iran

Abstract

An adaptive process consists of dynamic elements, and management rules which govern their run-time behaviors. The WS-CDL describes collaborative business processes between service consumers and providers. Adapting the processes to runtime changes becomes a demanding challenge, because the WS-CDL has static technology-dependent structure, and does not support the separation of concerns. Here, we propose a model-driven approach to transform WS-CDL into UML state machine (behavioral and protocol models), and subsequently into implementation code. Besides separating the business logic from the implementation, the state machine has a dynamic structure which is verifiable and adaptable. As a result, we can easily modify the process flow or change the management rules at run-time, and reflect their effects on the running process instances. We present an ‘itinerary purchase’ case study for prototyping the transformation rules and algorithm.

Keywords: Service Choreography, Model-driven transformation, Adaptation

Toward automatic Transformation of Service Choreography into UML State Machine

Choreography addresses the interaction that implements the collaboration between services. Choreography describes collaborative business processes (CBP) to achieve common goals among multiple distributed partners. It shows a global view of all interactions, and specifies the potential (observable) behaviors a partner can exhibit in order to interact. Orchestration refers to a composed business process which use both internal and external web services to fulfill its task (Bhuyan, P., Ray, A., & Mohapatra, D.P., 2015). Moreover, using service orchestration, each partner can provide its own internal realization of observable behaviors (Johann, E., & Tahamtan, A., 2008). Figure 1 shows a conceptual model of the CBP complementary concepts including service choreography, observable behavior, and service orchestration.

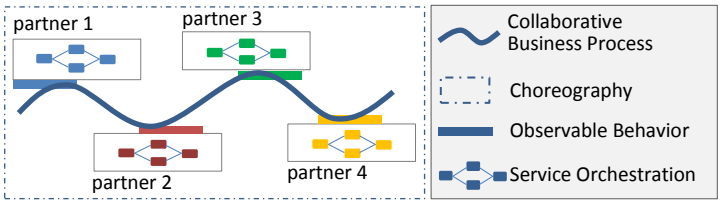

Figure 1. A conceptual model of a service-based CBP

The Web Service Choreography Description Language (WS-CDL) (Kavantzas et al, 2005) is the W3C recommended language for specifying service-based CBPs. An adaptive CBP consists of dynamic elements and management rules which govern their run-time behaviors (Li, Z., & Parashar, M., 2006). Since WS-CDL has a static structure, it is necessary to transform the service-based CBPs into adaptive models. Meanwhile, 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.

WS-CDL transformation is addressed in the literature, mostly with the goal of choreography verification. Here, we selected UML state machine (UML-SM) as an adaptive model to achieve the adaptation requirements of CBPs. The adaptation might be required in case of context (e.g., computational or environmental), or user preference, or business rule changes. In this regard, it is possible to transform both WS-CDL and WSBPEL documents into state-based models, and then integrate them by using nested property of UML-SM. In this paper, we only propose transformation of WS-CDL into UML-SM. As shown in Figure 2, first, a CBP is documented according to the WS-CDL specification. Then, the CBP document is transformed into behavioral and protocol state machines. Finally, implementation code is generated based on the state machines.

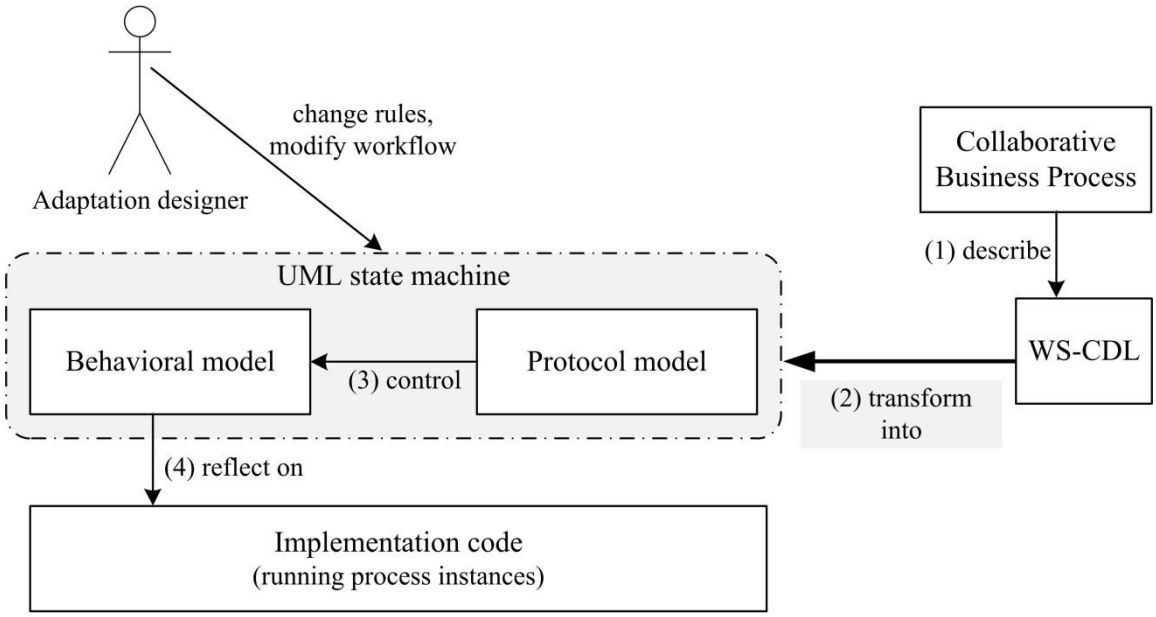

Figure 2. CBP adaptation model

The rest of the paper is organized as follows. We begin by explaining related studies and compare them regarding to adaptation issues. Next we describe an overview of WS-CDL specification. Then, we correspond WS-CDL with UML state machine and discuss about the rationale behind the transformation. After that, we demonstrate the transformation rules based on an itinerary purchase case study. The final section of the paper provides the conclusions.

Related Work

Mendling & Hafner (2008) propose a model driven transformation approach to drive BPEL process definitions from a global WS-CDL model. It proposes a mapping between WS-CDL and WSBPEL 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 (Weber, I., Haller, J., & Mulle, J.A., 2008) 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 (Decker, G., Kopp, O., Leymann, F., & Weske, M., 2007) 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 (Weiß, A., Karastoyanova, D., Molnar, D., & Schmauder, S., 2014).

More recently, a simple language CDL (Hongli et al., 2006) was 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 (Diaz et al., 2005), and Colored Petri-net (Valero et al., 2012) (Benabdelhafid et al., 2014) 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 point of view, we consider the below attributes to compare the choreography description languages/models. The comparison results are depicted in Table 1.

— Structure

- Dynamic. Dynamic structure means that the structure of a process must be flexible to being reconfigured and regulated dynamically in response to commands of management activities.
- Workflow support. It refers to supporting both workflow and services interaction patterns (e.g., sequence, parallel, synchronization, send, receive, etc.).
- Hierarchical (nested). A hierarchical process is designed level by level, for hiding unnecessary details at each abstraction level. At each level, there is a composite operation that may be broke down at the next lower level.
- Separation of concerns. Separation of concerns (McKinley, P.K., Sadjadi, M., Kasten, E.P., & Cheng, B., 2004) enables the separate development of the business logic and the cross-cutting concerns of a process (e.g., quality of service, implementation code).

— Management

- Manageable. A process is manageable, when its *structure* is reconfigured by, and its *runtime behaviors* are regulated by, management rules and protocol.

- Verifiable. Choreography verification consists of two main types of protocol mismatches. *Service interoperability* verification including message ordering and time constraints at design time (Benabdelhafid, M.S., & Boufaida, M., 2014). *Deadlock*, in which both parties are mutually waiting to receive some message from the other (Dumas, M., Benatallah, B., & Nezhad, H.R.M., 2008).

Table 1

Comparison of choreography modeling and description languages

		Dynamic Structure				Management	
Language / Model	Goal	Dynamic	Workflow support	Hierarchical	Separation of concerns	Manageable	Verifiable
WS-CDL	Specification	-	●	✔	-	-	-
WSCI	Specification	-	●	-	-	-	-
BPEL4Chor	Specification Execution	-	●	✔	-	-	-
CDL	Specification Verification	-	-	-	-	●	✔
UML state machine	Modeling Specification Verification	✔	✔	✔	✔	✔	✔
Timed automata	Verification	✔	●	-	-	●	✔
Colored Petri net	Verification	✔	✔	●	●	●	✔

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

An Overview of WS-CDL

WS-CDL is an XML-based language that describes peer-to-peer collaborations of participants by defining, from a global viewpoint, their common and complementary observable behavior; where ordered message exchanges result in accomplishing a common business goal. As shown in Figure 3, a choreography element contains activity, exception handling and finalizer parts.

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.

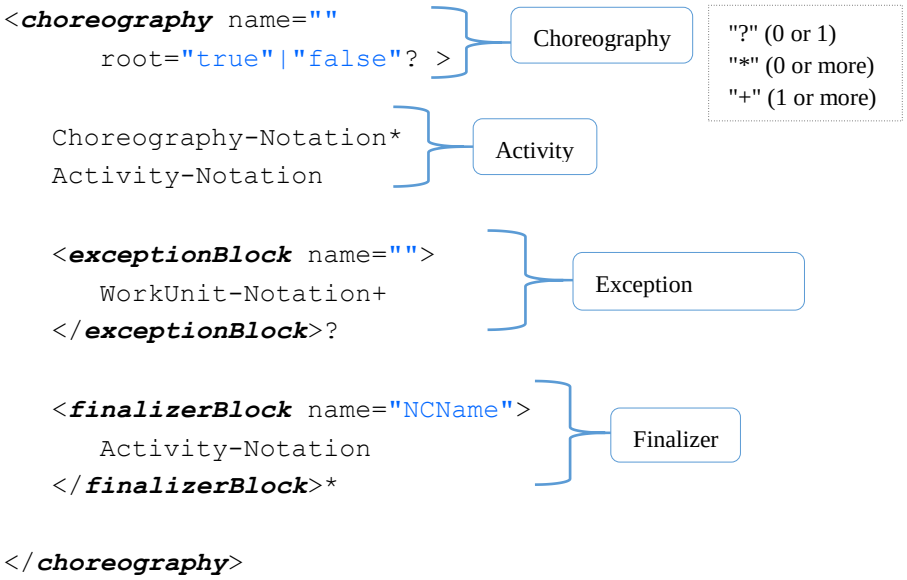

Figure 3. Structure of choreography element

Transformation

UML 2.4 proposes behavioral and protocol state machines. The behavioral state machine shows discrete behavior of a system through finite state transitions. The protocol state machine expresses the usage protocol of a system. The following nodes and edges are typical state machine elements: behavioral state, behavioral transition, protocol state, protocol transition, and different pseudo-states such as join, fork, entry/exit points, etc. The workflow and service interaction patterns are supported by UML state machine (Mellat et al., 2011).

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. Considering activities as states, two states are performed sequentially, when there is a transition from one state to another.
- **Parallel:** The activities within a parallel element are enabled concurrently. The parallel activity completes successfully, when all its enclosed activities complete successfully. The WS-CDL parallel element is modeled using the UML orthogonal regions. A state may be divided into orthogonal regions containing sub-states that execute concurrently and independently. The fork pseudo-state splits an incoming transition into two or more transitions entering the orthogonal regions. The join pseudo-state merges the transitions exiting from different orthogonal regions into one transition. As shown in Figure 4, we correspond a parallel element with a composite state, and consider a concurrent region for each activity within the parallel element. Since the parallel element does not have a name, the composite state is labeled with a temporary name.

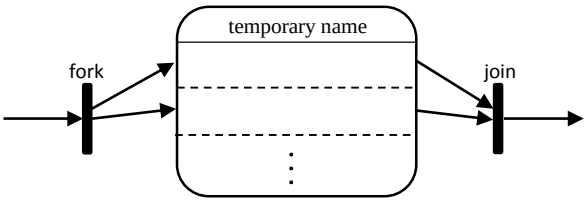

Figure 4. A composite state with orthogonal regions

– **Choice:** The choice ordering structure is similar to the UML choice pseudo-state. They both realize a dynamic conditional branch. Although the choice element encompasses one or more activities, only one activity is selected, and the other activities are disabled. The enclosed activities within a choice element are transformed into state-based elements subsequently. For example, Figure 5 is an equivalent state diagram for a WS-CDL choice element with three enclosed activities. Since each activity indicates a separate conditional branch, we transform the activity to the corresponding state-based elements.

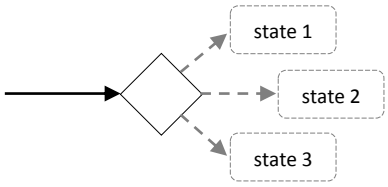

Figure 5. A choice pseudo-state

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. As shown in Figure 6, an interaction activity is transformed into a composite state (with hidden decomposition), and a transition entering the state.

  State: To represent a running interaction between participants, we consider a state, which is labeled with the interaction name.

  Trigger: When a message exchange with ‘action’ value equal to ‘requested’ is performed, its enclosing interaction is initiated. Therefore, we match the exception name with the transition trigger. The trigger specifies events that may induce state transition and also execution of actions.

  Guard: According to the WS-CDL specification, no attribute guard-condition is specified for an interaction. Therefore, we do not consider guard condition for the state transition. Nevertheless, to define condition expressions for an interaction, the interaction must be enclosed in a work-unit.

Action: Since an operation may be performed during an interaction, we correspond the operation with a transition's action. The action is executed when the transition is fired.

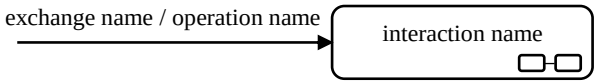

Figure 6. An interaction is corresponded with a composite state with hidden decomposition

— **No-action, Silent-action:** The no-action and silent-action activities are used, when a participant does not perform any action, or perform an action without any observable operational details, respectively. The no-action specifies a 'waiting' state for its enclosing choreography. In other words, the choreography does not perform any action, while it is waiting for an expecting event to continue. Similarly, the silent-action specifies a choreography, which is waiting for hidden operations to be completed, and then continue the performance. Consequently, as shown in Figure 7, we consider a basic state (with no action) for a no-action activity, and a composite state (with hidden decomposition) for a silent-action activity.

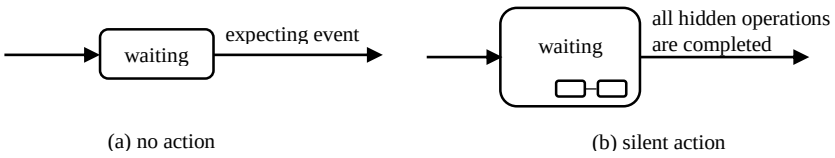

Figure 7. The 'no action' / 'silent action' activities are transformed to 'basic state' / 'composite state with hidden decomposition', respectively

— **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. Similarly, a composite state can include other composite or basic states in a nested way. Therefore, we correspond a perform activity with a composite

state. As shown in Figure 8, the composite state is labeled with the value of the 'name' attribute. Since the performed choreography encloses activities independently, the transformation algorithm must continue recursively to convert all enclosed activities to states and display them within/inside the (enclosing/parent) composite state.

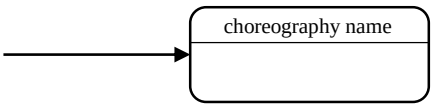

Figure 8. A choreography element is corresponded with a composite state

- **Exception block, Finalizer block:** We described the exception handling and finalizer blocks (see previous section, An Overview of WS-CDL, p. 8). The exception block contains one or more work-units, each work-unit handle an exceptional circumstance. The finalizer block contains required activities for finalizing its enclosing choreography performance. These blocks are simply transformed to composite states which have sub-states to cover work-units or activities. As shown in Figure 9, the composite state is labeled with the block name.

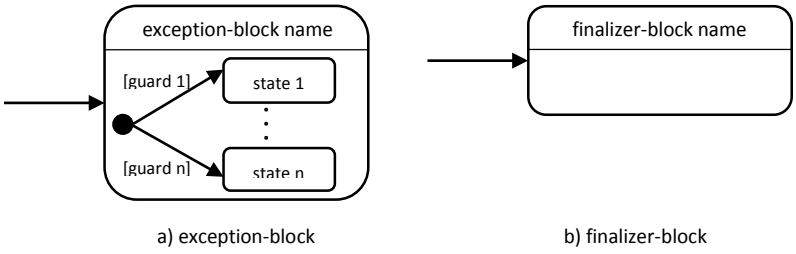

Figure 9. The exception and finalizer blocks are corresponded to composite states. Nested states should be considered subsequently for the activities within the blocks

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. Clearly, a work-unit is equal to a composite state and an entering transition with a guard condition. As shown in Figure 10, the composite state is labeled with the work-unit name. It also has sub-states corresponding to the work-unit activities. Each entering transition has a guard condition similar to the work-unit guard condition.

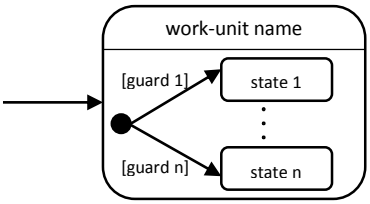

Figure 10. A work-unit is corresponded with a composite state. Nested states should be considered subsequently for the activities within the work-unit

To sum up, in this section, we described the WS-CDL choreography elements, and tried to model each element’s behavior through finite state transitions. In this way, we transformed ‘choreography elements and attributes’ into ‘state-transition and event-condition-action’. Table 2 summarizes the transformation rules.

Table 2

WS-CDL to UML state machine mapping table

WS-CDL	UML state machine	Label	Event [Guard]/Action
Root choreography	Initial state and composite state	Choreography name	-
Enclosed choreography	Initial state	-	-
</Choreography>	Final state	-	-
Interaction	Composite state with hidden decomposition	Interaction name	Exchange name [null] / operation name
Perform	Composite state	Choreography name	-
No-action	Basic state	Waiting	Exception name [null] / null
Silent-action	Basic state	Waiting	-
Sequence	Transitions	-	-
Parallel	Fork - Orthogonal regions - Join	Temporary name	-
Choice	Choice	-	-
Finalizer block	Composite state	Finalizer block name	-
Exception block	Composite state	Exception block name	-
Work unit	Composite state	Work unit name	Null [guard name] / null

Case study

Here, we adopt and extend the ‘itinerary purchase’ example (Order Processing Center Application, 2012) (Douglas, 2013) 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. The itinerary purchase scenario is as follows. First, the customer requests the travel agency for available itineraries, and then the travel agency sends all available itineraries to the customer. Next, the customer selects de-

sired itinerary and requests the travel agency for reservation. The travel agency starts two parallel choreographies with the hotel and airline parties, and waits until reservation responses arrive. If both of reservations are done, then the travel agency calculates total cost of itinerary locally (indeed, it calculates the airline, travel agency, hotel and other commissions plus the base costs). 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. Finally a choreography is started to notify the customer about the payment and itinerary information. The choreographies of the mentioned ‘itinerary purchase’ CBP is shown in Figure 11.

```
<choreography name="itineraryPurchase" root="true">
  <sequence>
    <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>
    <perform choreographyName="itineraryReservation"></perform>
    <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">
      <perform choreographyName="successNotification"></perform>
    </workunit>
  </finalizerBlock>
</choreography>

<choreography name="itineraryReservation">
  <parallel>
    <perform choreographyName="flightReservation"></perform>
    <perform choreographyName="roomReservation"></perform>
  </parallel>
</choreography>
```

Figure 11. 'Itinerary purchase' state machine

According to the transformation rules that we mentioned in previous section, we present a transformation algorithm. A pseudo code of the transformation algorithm is shown in the paper appendix. We applied the algorithm to the above itinerary purchase choreography to get its corresponding state machine. As a result, Figure 12 and Figure 13

show the equivalent state machine of the above choreography and its exception block respectively.

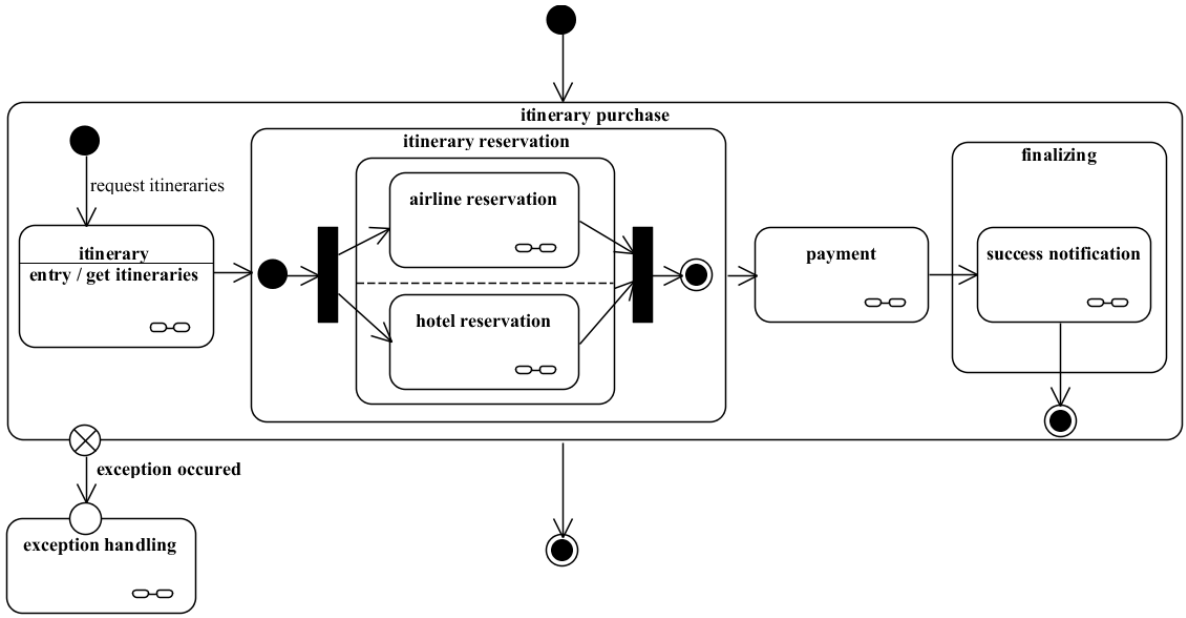

Figure 12. 'Itinerary purchase' state machine

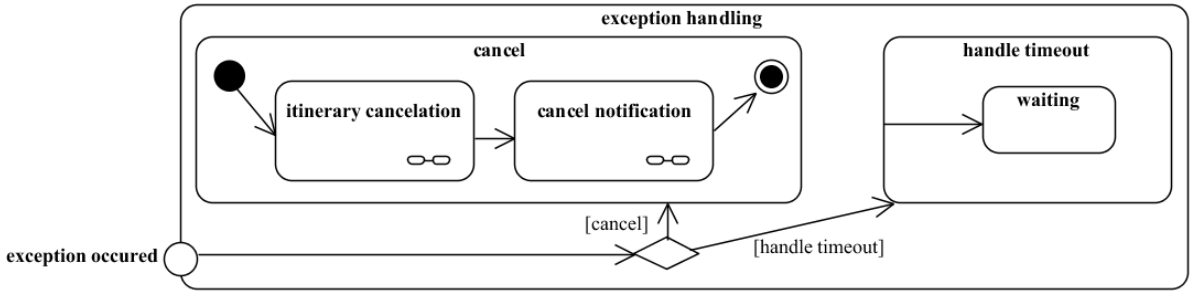

Figure 13. 'Exception handling' composite state

Semantic Preservation

Providing proof techniques for showing full semantic preservation of model transformation is a very difficult problem, on which there has been little work so far (Hülsbusch et al., 2010). Instead of generally proving total correctness of a transformation, a number of approaches, carry out run-time checks of equivalence between a given source and generated target model, that is, partial correctness. Here we describe semantic preservation of state machine models in order to prove 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. UML-SM). In the following list, we show that our proposed transformation rules preserve ordering of messages and semantic of interactions.

- **Ordering and Composing Structures.** WS-CDL's ordering and composing structures are corresponded with state based elements in a straightforward form (see the Transformation section).
- **Flow Control.** To control flow of messages, WS-CDL uses guard conditions in Exception Block and Work Unit. Similarly, UML-SM controls flow of messages by evaluating guards associated with transitions.
- **Interaction.** Choreography interaction includes service invocation between two partners in which one partner requests for an operation, and the other partner executes the operation and reply. Here we show how to realize service invocation between two partners using UML-SM. In particular, we expand 'interaction composite state' shown in Figure 6, to expose its enclosed states, transitions and series of events. As shown in Figure 14, a

consumer enters ‘waiting for response’ state while sending a request event to provider. The request event is labeled with operation name and induces the provider to exit ‘waiting for request’ state and execute requested state machine. Upon receipt of response event, the consumer exits from ‘waiting for response’ state and continue its state machine.

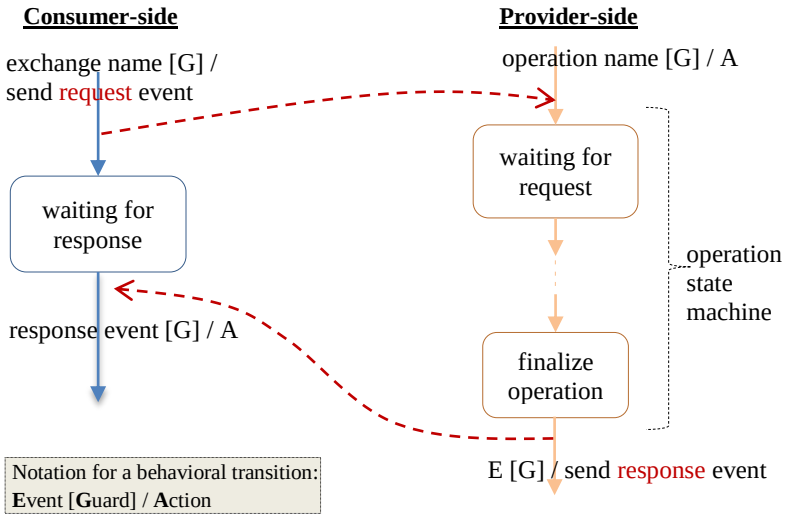

Figure 14. A realization of service invocation using UML-SM (expanded version of Figure 6)

Conclusion

In this paper, first, we studied the required characteristics of an adaptive service-based CBP such as dynamicity, verifiability, manageability, etc. As a result, we selected UML state machine for modeling adaptive CBPs. While the behavioral state machine shows the runtime behaviors of a CBP, the protocol state machine controls the actions.

Next, we proposed a model-driven approach to transform the WS-CDL specification into UML state machine automatically. We presented the transformation rules and the rationale behind each rule. The benefits of the transformation include:

- The adaptation strategies like reconfiguration, reselection are easily realized by modifying the behavioral and protocol state machines. For example, by adding / removing / merging / replacing states and transitions.
- We can suspend the failed instance of a process for adaptation, and then, resume it without interrupting other process instances.
- We can separately develop the business process and policies, and reflect their effects on the running process instances.

In future, we will develop a software tool that implements the pseudo-code of transformation algorithm. We will also propose the transformation of WSBPEL (i.e., an orchestration language) into UML state machine. Then, we can transform both WS-CDL and WSBPEL into their corresponding state machines, and integrate them in a nested way. According to the reflective-state design pattern (Ferreira, L.L., & Rubira, M.F., 1998), we will deploy the state machines on meta-level, and their implementation on base-level. We will consider concrete states and services to realize the functionalities that are defined at meta-level. Consequently, the adaptation designer (or an automatic adaptation unit) can easily modify meta-level which mirrors the behaviors of each process instance distinctly.

Appendix: A Pseudo-code of Transformation Algorithm

```
#=== ABBREVIATIONS ===
/* 'CS' (Composite State), 'HCS' (a Composite State that is
 * shown with Hidden decomposition), 'BS' (Basic State),
 * 'F' (Fork), 'J' (Join)
 * 'IS', (Initial State), 'FS': (Final State)
 */

#=== START TRANSFORMATION ===
SET query to '/package/choreography/[@root=true]'
CALL findTag with query
CALL TRANSFORM

#=== SUB-MODULES ===
string FUNCTION readTag ()
  Extract the tag referred by read-pointer
  Update read-pointer to next tag

void FUNCTION findTag (query)
  Find the tag according to query
  Update read-pointer

void FUNCTION draw (object, name, event, guard, action)
  Find the tag according to query
  Update read-pointer

#=== MAIN MODULE ===
void FUNCTION TRANSFORM ()
CALL readTag RETURNING tag

IF tag EQUAL 'choreography' THEN
  CALL draw with 'IS'
  CALL TRANSFORM
ENDIF
IF tag EQUAL '</choreography>' THEN
  CALL draw with 'FS'
  RETURN
ENDIF
IF input EQUAL tag THEN
  RETURN '</'+tag+'>'
ENDIF

/* basic activities */
CASE tag OF
  'perform':
    IF nested states must be shown THEN
```

```
       SAVE read-pointer
       SET query to
'/package/choreography/[@name=perform.choreographyName]'
       CALL findTag with query
       CALL draw with 'CS'
       CALL TRANSFORM
       RESUME read-pointer
     ELSE
       CALL draw with 'HCS'
     ENDIF
  'interaction':
     CALL draw with 'HCS'
  'noaction':
     CALL draw with 'BS'
  'silentaction':
     CALL draw with 'BS'
ENDCASE

/* ordering structures */
IF tag EQUAL 'sequence' OR 'parallel' OR 'choice' THEN
  IF tag EQUAL 'parallel' THEN
     CALL draw with 'F'
  ENDIF
IF tag EQUAL 'choice' THEN
     CALL draw with 'C'
  ENDIF
  REPEAT
     CALL TRANSFORM RETURNING output
  UNTIL output NOT EQUAL '</'+tag+'>'
  IF tag EQUAL 'parallel' THEN
     CALL draw with 'J'
  ENDIF
ENDIF
```

References

Order Processing Center Application. (2012). Retrieved from https://wiki.sei.cmu.edu/sad/index.php/OPC_C%26C_View

Benabdelhafid, M.S., & Boufaida, M. (2014). Toward a Better Interoperability of Enterprise Information Systems: A CPNs and Timed CPNs -based Web Service Interoperability Verification in a Choreography. *Procedia Technology, 16*, 269–278.

Benabdelhafid, M.S., Bérard, B., & Boufaida, M. (2014). Analyzing Behavioral Compatibility for Web Service Choreography Using Colored Petri Nets and ASK-CTL. *The Sixth International Conferences on Advanced Service Computing*, (pp. 32-39).

Bhuyan, P., Ray, A., & Mohapatra, D.P. (2015). A Service-Oriented Architecture (SOA) Framework Component for Verification of Choreography. *Computational Intelligence in Data Mining, 3*, 25-35.

Decker, G., Kopp, O., Leymann, F., & Weske, M. (2007). BPEL4Chor: Extending BPEL for modeling choreographies. *IEEE International Conference on Web Services*, (pp. 296-303).

Diaz, G., Pardo, J., Cambronero, M., Valero, V., & Cuartero, F. (2005). Automatic translation of WS-CDL choreographies to timed automata. *Formal Techniques for Computer Systems and Business Processes* (pp. 230-242). Springer Berlin Heidelberg.

Douglas, A. (2013). *WS-CDL Eclipse*. Retrieved from http://sourceforge.net/projects/wscdl-eclipse/

Dumas, M., Benatallah, B., & Nezhad, H.R.M. (2008). Web service protocols: Compatibility and adaptation. *IEEE Data Eng. Bull, 31*(3), 40-44.

Ferreira, L.L., & Rubira, M.F. (1998). The Reflective State Pattern. *Proceedings of the Pattern Languages of Program Design, TR-WUCS-98-25.* Monticello, Illinois-USA.

Hongli, Y., Xiangpeng, Z., Zongyan, Q., Geguang, P., & Shuling, W. (2006). A formal model for web service choreography description language (WS-CDL). *ICWS.*

Hülsbusch, M., König, B., Rensink, A., Semenyak, M., Soltenborn, C., & Wehrheim, H. (2010). Showing full semantics preservation in model transformation-a comparison of techniques. *Integrated Formal Methods*, 183-198.

Johann, E., & Tahamtan, A. (2008). Temporal conformance of federated choreographies. *Database and Expert Systems Applications*, 668-675.

Kavantzas, N., Burdett, D., Ritzinger, G., Fletcher, T., Lafon, Y., & Barreto, C. (2005). Web Services Choreography Description Language Version 1.0. *W3C WS-CDL Working Group*. Retrieved from http://www.w3.org/TR/ws-cdl-10/

Li, Z., & Parashar, M. (2006). Enabling Dynamic Composition and Coordination for Autonomic Grid Applications using the Rudder Agent Framework. *The Knowledge Engineering Review, Vol. 00:0*, 1-15.

McKinley, P.K., Sadjadi, M., Kasten, E.P., & Cheng, B. (2004). *A taxonomy of compositional adaptation.* MSU-CSE-04-17.

Mellat, A., Nematbakhsh, N., Farahi, A., & Mardukhi, F. (2011). Suitability of UML state machine for modeling choreography of services. *International Journal of Web & Semantic Technology (IJWesT), Vol.2, No.4*.

Mendling, J. & Hafner, M. (2005). From inter-organizational workflows to process execution: Generating BPEL from WS-CDL. *On the Move to Meaningful Internet Systems 2005: OTM 2005 Workshops.* Springer Berlin Heidelberg.

Mendling, J. & Hafner, M. (2008). From WS-CDL choreography to BPEL process orchestration. *Journal of Enterprise Information Management, 21*(5), 525-542.

Valero, V., Macià, H., Pardo, J.J., Cambronero, M.E., & Díaz, G. (2012). Transforming Web Services Choreographies with priorities and time constraints into prioritized-time colored Petri nets. *Science of Computer Programming, 77*(3), 290-313.

Weber, I., Haller, J., & Mulle, J.A. (2008). Automated derivation of executable business processes from choreographies in virtual organisations. *International Journal of Business Process Integration and Management, 3*(2), 85-95.

Weiß, A., Karastoyanova, D., Molnar, D., & Schmauder, S. (2014). Coupling of existing simulations using bottom-up modeling of choreographies. *In Workshop on simulation technology: systems for data intensive simulations (SimTech@ GI) in conjunction with INFORMATIK*, (pp. 101-112).