APPENDIX A: VISUAL EDITOR FOR FLORA-2 BASED SWS SPECIFICATIONS (VSCHOR)

Visual Semantic Choreography (VSChor) is a visual software environment which we developed to facilitate designing and deploying goals and web services with Flora-2 specifications. Choreography designers can define concepts, frames, predicates and specify the structures of goals and web services by filling-out the prepared forms and automatically generate Flora-2 specifications that are ready to run on the developed choreography engine. The software also embeds the engine code and related libraries in a deployment folder. The user can test the choreography by just running a single batch file, provided the Flora-2 system is already installed on the local platform.

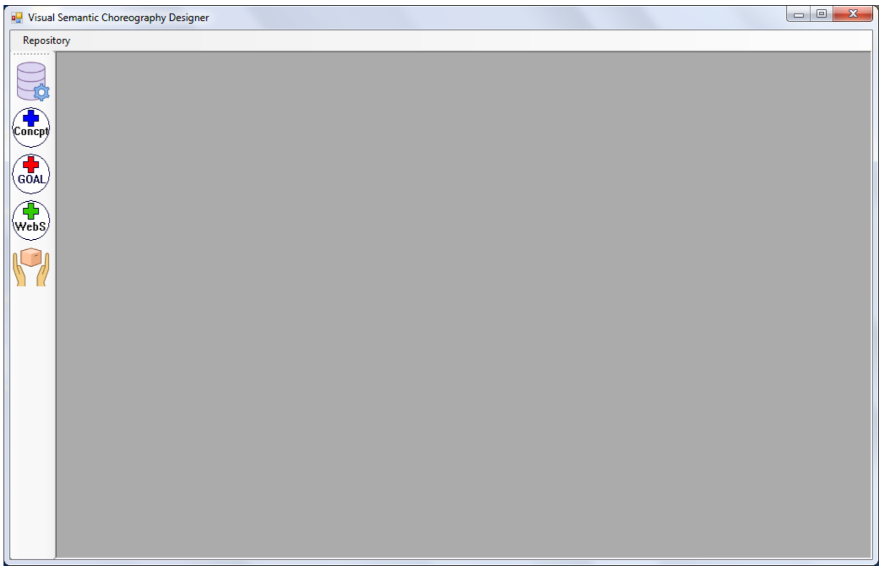
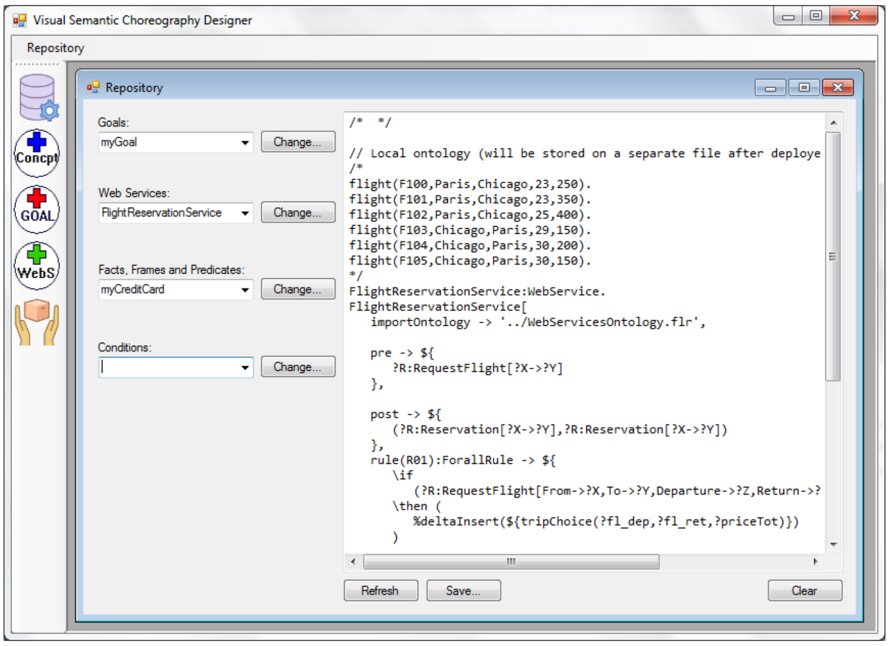


Figure 1. Main entrance form of VSChor

The main window of the program is shown in Figure 1. The left tool box contains five icons: from top to down, Repository, Add Concept, Add Goal, Add Web Service, and Deploy, respectively. The repository button is used to show the currently registered concepts, frames, predicates, goals, and web services (Figure 2). It also allows the user to change the definitions of goals and web services.



Goals: myGoal

Web Services: FlightReservationService

Facts, Frames and Predicates: myCreditCard

Conditions:

```
/*  */

// Local ontology (will be stored on a separate file after deploye
/*
flight(F100,Paris,Chicago,23,250).
flight(F101,Paris,Chicago,23,350).
flight(F102,Paris,Chicago,25,400).
flight(F103,Chicago,Paris,29,150).
flight(F104,Chicago,Paris,30,200).
flight(F105,Chicago,Paris,30,150).
*/
FlightReservationService:WebService.
FlightReservationService[
   importOntology -> '../WebServicesOntology.flr',

   pre -> ${
      ?R:RequestFlight[?X->?Y]
   },

   post -> ${
      (?R:Reservation[?X->?Y],?R:Reservation[?X->?Y])
   },
   rule(R01):ForallRule -> ${
      \if
         (?R:RequestFlight[From->?X,To->?Y,Departure->?Z,Return->?
      \then (
         %deltaInsert(${tripChoice(?fl_dep,?fl_ret,?priceTot)})
      )
```

Figure 2. Repository form

Repositories can be saved and loaded by the Repository menu located at the top-left corner of the main form (Figure 3).

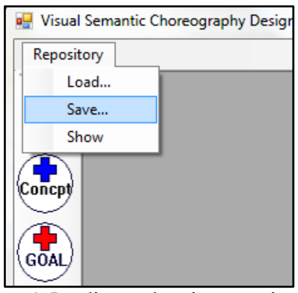
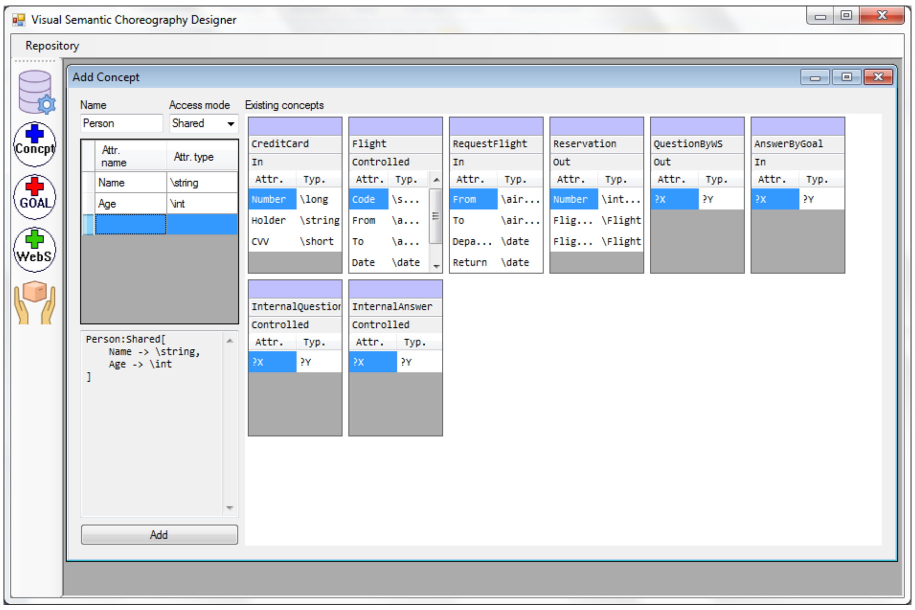


Figure 3. Loading and saving repositories

Concepts can be defined by Add Concept form (Figure 4). The user can enter concept attributes and their types by filling out the available grid and determine the proper mode type by using the available combobox.



Name: Person
Access mode: Shared

Attr. name	Attr. type
Name	\string
Age	\int

```
Person:Shared[
    Name -> \string,
    Age -> \int
]
```

CreditCard	
In	
Attr.	Typ.
Number	\long
Holder	\string
CVV	\short

Flight	
Controlled	
Attr.	Typ.
Code	\s...
From	\a...
To	\a...
Date	\date

RequestFlight	
In	
Attr.	Typ.
From	\air...
To	\air...
Depa...	\date
Return	\date

Reservation	
Out	
Attr.	Typ.
Number	\int...
Flig...	\Flight
Flig...	\Flight

QuestionByWS	
Out	
Attr.	Typ.
?X	?Y

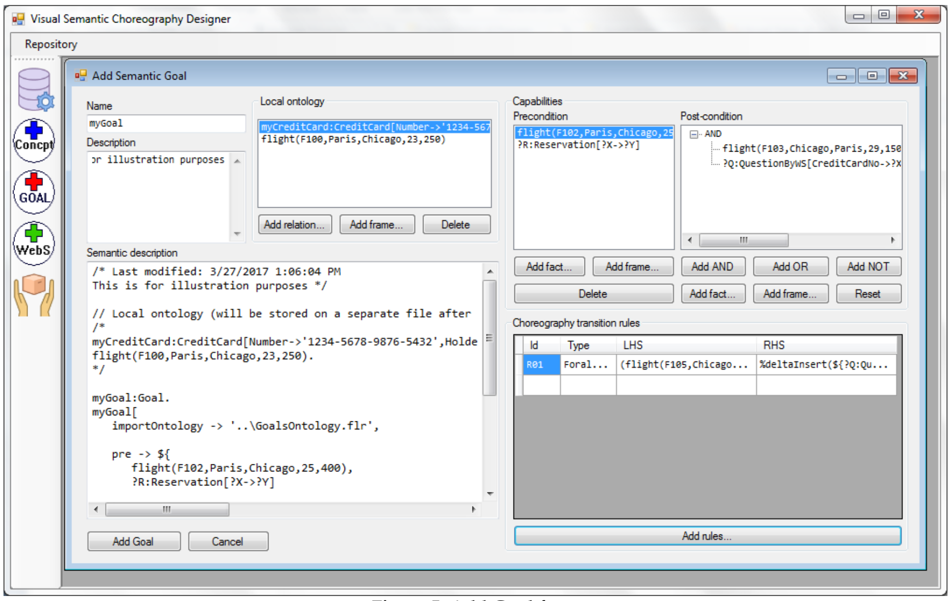
AnswerByGoal	
In	
Attr.	Typ.
?X	?Y

InternalQuestion	
Controlled	
Attr.	Typ.
?X	?Y

InternalAnswer	
Controlled	
Attr.	Typ.
?X	?Y

Figure 4. Add Concept form

Goals and web services can use instances of the concepts in their specifications. The Add Goal and Add Service forms are essentially the same, so here we only illustrate Add Goal form, depicted in Figure 5.



Name: myGoal
Description: ɔr illustration purposes

Local ontology:

```
myCreditCard:CreditCard[Number->'1234-567
flight(F100,Paris,Chicago,23,250)
```

Semantic description:

```
/* Last modified: 3/27/2017 1:06:04 PM
This is for illustration purposes */

// Local ontology (will be stored on a separate file after
/*
myCreditCard:CreditCard[Number->'1234-5678-9876-5432',Holde
flight(F100,Paris,Chicago,23,250).
*/

myGoal:Goal.
myGoal[
   importOntology -> '..\GoalsOntology.flr',

   pre -> ${
      flight(F102,Paris,Chicago,25,400),
      ?R:Reservation[?X->?Y]
```

Precondition:

```
flight(F102,Paris,Chicago,25
?R:Reservation[?X->?Y]
```

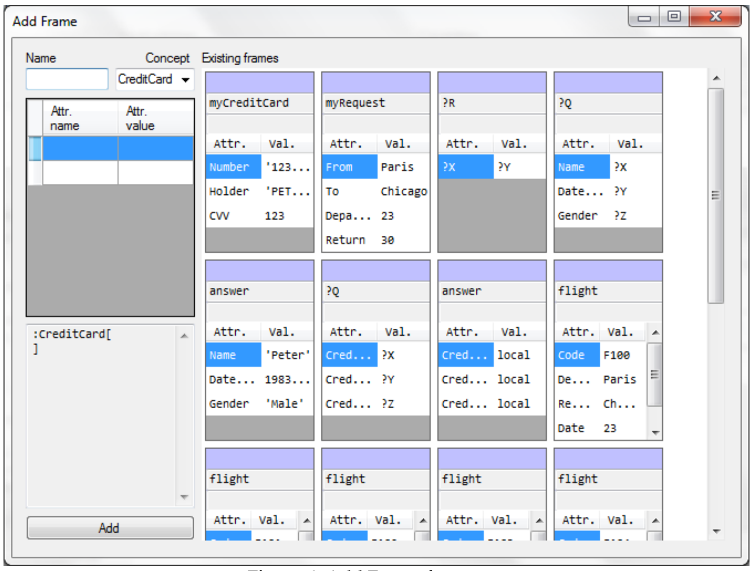
Post-condition:

```
AND
   flight(F103,Chicago,Paris,29,150
   ?Q:QuestionByWS[CreditCardNo->?X
```

Id	Type	LHS	RHS
R01	Foral...	(flight(F105,Chicago...	%deltaInsert(${?Q:Qu...

Figure 5. Add Goal form

The Add Goal form contains places for the name, description, local ontology, capabilities, pre and post conditions, and transition rules. The Semantic description box shows the current textual description of the goal in Flora-2. The Add relation and Add frame buttons let the user define new (or use already defined) relations and rules respectively (Figure 6).



Concept: CreditCard

```
:CreditCard[
]
```

myCreditCard	
Attr.	Val.
Number	'123...
Holder	'PET...
CVV	123

myRequest	
Attr.	Val.
From	Paris
To	Chicago
Depa...	23
Return	30

?R	
Attr.	Val.
?X	?Y

?Q	
Attr.	Val.
Name	?X
Date...	?Y
Gender	?Z

answer	
Attr.	Val.
Name	'Peter'
Date...	1983...
Gender	'Male'

?Q	
Attr.	Val.
Cred...	?X
Cred...	?Y
Cred...	?Z

answer	
Attr.	Val.
Cred...	local
Cred...	local
Cred...	local

flight	
Attr.	Val.
Code	F100
De...	Paris
Re...	Ch...
Date	23

flight	
Attr.	Val.

flight	
Attr.	Val.

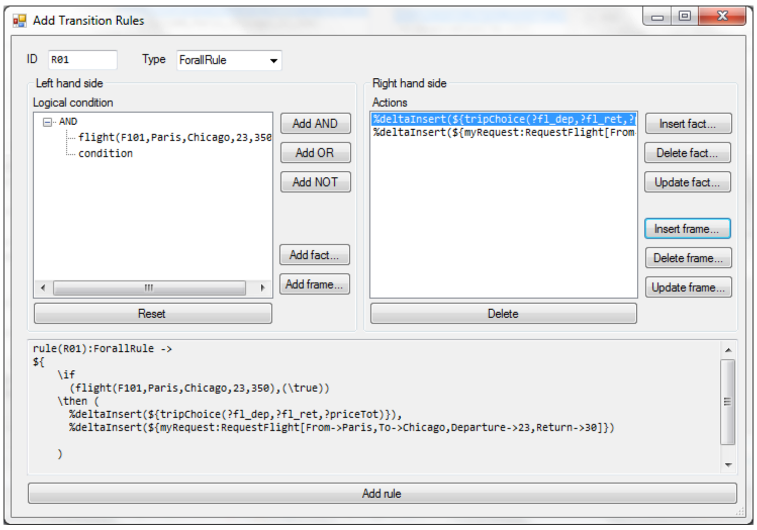
flight	
Attr.	Val.

flight	
Attr.	Val.

Figure 6. Add Frame form

Post-condition of a goal may contain a complex logic expression consisting of the *and*, *or*, and *not* operators, as well as frames and predicates. VSChor lets the user create a logic tree representing the logic expression of the goal post-condition (Figure 5. Post-condition view).

Transition rules are created through another form (Figure 7), accessible by clicking on the Add rule button on Add Goal and Add Web Service forms. The user can specify the name, rule type (either Forall or Choose), antecedent (left-hand side), and consequent (right-hand side) of the rules. Similar to the goal post-condition, the antecedent can contain a logic expression of any complexity. The consequent can consist of a set of actions, which should be one of `%deltaInsert`, `%deltaDelete`, or `%deltaUpdate`.



ID: R01 Type: ForallRule

```
AND
  flight(F101,Paris,Chicago,23,350
  condition
```

```
%deltaInsert(${tripChoice(?fl_dep,?fl_ret,?
%deltaInsert(${myRequest:RequestFlight[From
```

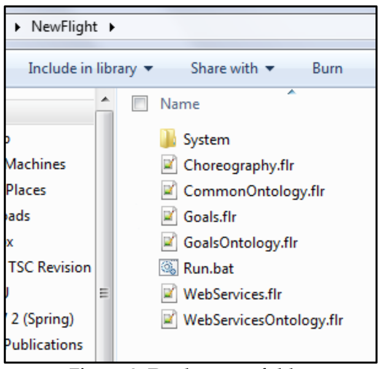
```
rule(R01):ForallRule ->
${
    \if
      (flight(F101,Paris,Chicago,23,350),(\true))
    \then (
      %deltaInsert(${tripChoice(?fl_dep,?fl_ret,?priceTot)}),
      %deltaInsert(${myRequest:RequestFlight[From->Paris,To->Chicago,Departure->23,Return->30]})

    )
```

Figure 7. Add transition rule form

At any stage the user can modify the contents of goals (web services) using the *change semantic goal (web service)* form, accessible through the repository form.

After the choreography design is completed, a deployment package is generated by clicking on the deployment button (Figure 1. last button). The deployment package contains all the files related to goals, web services, common ontology, choreography engine, function libraries, as well as a batch file called "Run.bat" (Figure 8). The batch file allows the user to test the choreography simply by double clicking on it.



System
Choreography.flr
CommonOntology.flr
Goals.flr
GoalsOntology.flr
Run.bat
WebServices.flr
WebServicesOntology.flr

Figure 8. Deployment folder

APPENDIX B: E-BNF GRAMMAR FOR FLORA-2 GOAL AND WEB SERVICE SPECIFICATIONS

Here, we present the current grammar of goal and web service specifications written in E-BNF [26] (Table 1).

Table 1. E-BNF grammar for goal and web service specifications

<webservice>	`::= <oid> : WebService '['`
	`importOntology -> <symbolString> ,`
	`capability -> $ '{'`
	`pre -> $ '{' <condition> '}' ,`
	`post -> $ '{' <andcondition> '}'`
	`'}'`
	`[, <wstransitionrules> ] ']' .`

<goal>	`::= <oid> : Goal '['`
	`importOntology -> <symbolString> ,`
	`capability -> $ '{'`
	`pre -> $ '{' <andcondition> '}' ,`
	`post -> $ '{' <condition> '}'`
	`'}'`
	`[, <gtransitionrules> ] ']' .`

<andcondition>	`::= ( <frame> | <predicate> ) {, ( <frame> | <predicate> ) }`
<condition>	`::= <condition> ; <term> | <term>`
<term>	`::= <term> , <factor> | <factor>`
<factor>	`::= <frame> | <predicate> | '(' <condition> ')' | '(' \+ <condition> ')'`
<wstransitionrules>	`::= <wsrule> {, <wsrule> }`
<gtransitionrules>	`::= <grule> {, <grule> }`

<wsrule>	`::= wsRule ( <oid> ) : <ruletype> -> $ '{'`
	`\if <condition>`
	`\then '(' <actions> ')'`
	`[ \else '(' <actions> ')' ] '}'`

<grule>	`::= gRule ( <oid> ) : <ruletype> -> $ '{'`
	`\if <condition>`
	`\then '(' <actions> ') '`
	`[ \else '(' <actions> ')' ] '}'`

<ruletype>	`::= ForallRule | ChooseRule`
<actions>	`::= <action> { , <action> }`

<action>	`::= %deltaInsert '(' <frame> ')' |`
	`%deltaInsert '(' <predicate> ')' |`
	`%deltaDelete '(' <restrictedframe> ')' |`
	`%deltaDelete '(' <predicate> ')' |`
	`%deltaUpdate '(' <objname> , <attrname> , <f_term> , <f_term> ')'`

<predicate>	`::= <predname> [ '(' [ <f_term> {, <f_term> } ] ')' ]`
<frame>	`::= <objname> : <concept> '[' [ <attribute-value-pairs> ] ']'`
<attribute-value-pairs>	`::= <attrname> -> <f_term> { , <attrname> -> <f_term> }`
<restrictedframe>	`::= <objname> '[' [ <attribute-value-pairs> ] ']'`
<concept>	`::= <oid>`
<attrname>	`::= <oid>`
<objname>	`::= <oid>`
<predname>	`::= <oid>`
<symbolString>	`::= '''<string>'''`
<string>	::= *Any sequence of characters*
<oid>	::= *Any valid Flora-2 object identifier*
<f_term>	::= *Any valid Flora-2 term*

APPENDIX C: MORE CHOREOGRAPHY EXAMPLES

1 Semantic authentication

Service requesters usually are asked to be authenticated by online services before being able to utilize them. Authentication information can be asked at the beginning, in the middle or in last steps of the service consumption process. Here we present a semantic choreography for the authentication process. If the service requester (goal) and service provider (web service) use the same terminology, the requester can pass authentication information in a choreography to the web service. Listings 1 and 2 depict the semantic choreography specifications of the goal and the web service respectively.

Listing 1. Goal choreography specification for semantic authentication

```
// Local ontology (will be stored on a separate file after deployed)
/*
myUserName('Riccardo').
myPassword('98765').
*/
myGoal:Goal.
myGoal[
   importOntology -> '../Auth/GoalsOntology.flr',

   capability -> ${
        pre -> ${
           ar:AuthenticationRequest[UserName->?UN, Password->?PW],
           AuthenticationRequest(UserName,?UN),
           AuthenticationRequest(Password,?PW)
        },

        post -> ${
           ?AV:AuthenticationValidation[?X->?Y] \or
           AuthenticationValidation() \or
           AuthenticationValidation(?Z) \or
           AuthenticationValidation(?W,?S)
        }
   },
   gRule(R01):ForallRule -> ${
      \if (
         ?X:QuestionByWS[UserName->?Y]@WM,
          myUserName(?UN)@WM
      )
      \then (
        %deltaInsert(${A:AnswerByGoal[UserName->?UN]})
      )
   },
   gRule(R02):ForallRule -> ${
      \if (
         ?X:QuestionByWS[Password->?Y]@WM,
          myPassword(?PW)@WM
      )
      \then (
        %deltaInsert(${A:AnswerByGoal[Password->?PW]})
      )
   },
   gRule(R03):ForallRule -> ${
      \if (
         QuestionByWS(UserName,?X)@WM,
          myUserName(?UN)@WM
      )
      \then (
        %deltaInsert(${AnswerByGoal(UserName,?UN)})
      )
   },
   gRule(R04):ForallRule -> ${
      \if (
         QuestionByWS(Password,?X)@WM,
          myPassword(?PW)@WM
      )
      \then (
        %deltaInsert(${AnswerByGoal(Password,?PW)})
      )
   }
].
```

Listing 2. Web Service choreography specification of semantic authentication

```
// Local ontology (will be stored on a separate file after deployed)
/*
DB_UserName_Password('PeterJM','12345').
DB_UserName_Password('Angel83','112233').
DB_UserName_Password('Riccardo','98765').
DB_UserName_Password('Agent007','7777777').
*/
AuthenticationService:WebService.
AuthenticationService[
   importOntology -> '../Auth/WebServicesOntology.flr',

   capability -> ${
        pre -> ${
           ?AR:AuthenticationRequest[?X->?Y]
        },

        post -> ${
           ?AV:AuthenticationValidation[?X->?Y]
        }
   },
   wsRule(R01):ForallRule -> ${
      \if (
         (?AR:AuthenticationRequest[?X->?Y])@WM
      )
      \then (
         %deltaInsert(${Q:QuestionByWS[UserName->_]}),
         %deltaInsert(${Q:QuestionByWS[Password->_]})
      )
   },
   wsRule(R02):ForallRule -> ${
      \if (
         (?X:AnswerByGoal[UserName->?UN])@WM,
         (?Y:AnswerByGoal[Password->?PW])@WM,
          DB_UserName_Password(?UN,?PW)@WM
      )
      \then (
         %deltaInsert(${av:AuthenticationValidation[
                           UserName->?UN, Password->?PW]})
      )
   },
   wsRule(R03):ForallRule -> ${
      \if (
         (?X:AnswerByGoal[UserName->?UN])@WM,
         (?Y:AnswerByGoal[Password->?PW])@WM,
          (\+ DB_UserName_Password(?UN,?PW))@WM
      )
      \then (
         %deltaInsert(${M:Message[
                          WS->'Incorrect username or password.']})
      )
   }
].
```

In this example, precondition of the goal contains two presentations of the request: one in the form of the frame `ar:AuthenticationRequest[UserName->?UN, Password->?PW]` and one in the form of the predicate pair { `AuthenticationRequest(UserName,?UN)`, `AuthenticationRequest(Password,?PW)`} . Also, the post-condition can be satisfied in different ways. On the other side, the web service only understands the requests in the form of frame. Similar to Flight Reservation Service, the choreography is progressed through a semantic conversion between the goal and web service.

2 Flight reservation with data granularity mismatch

Here we show that the choreography specifications of the goal and web service used in Flight Reservation Service can have different data granularity. As Flora-2 keeps the frames in the form of separated `attribute->value` pairs, this mismatch is handled by the choreography engine. Listings 3 and 4 contain the specifications of the goal and web service respectively.

Listing 3. Goal semantic choreography specification for flight reservation

```
// Local ontology (will be stored on a separate file after deployed)
/*
Name('Peter').
DateOfBirth(19830622).
Gender('Male').
CreditCardNo('1234432156788765').
CreditCardHolder('PETER JACKSON').
CreditCardCVV(123).
*/
myGoal:Goal.
myGoal[
   importOntology -> '../Flight/GoalsOntology.flr',
   capability -> ${
       pre -> ${
          myRequest:RequestFlight[
              From->'Paris',
              To->'Chicago',
              Departure->23,
              Return->30]
       },

       post -> ${
          (?R:Reservation[?X->?Y])
       }
   },
   gRule(R01):ChooseRule -> ${
      \if
         (tripChoice(?fl_dep,?fl_ret,?P), (\+ trip:Trip))@WM
      \then ( %deltaInsert(${trip:Trip[Dep->?fl_dep,Ret->?fl_ret]}) )
   },

   gRule(R02):ForallRule -> ${
      \if (
         (?Q:QuestionByWS[Name->?X], Name(?N))@WM
      )
      \then ( %deltaInsert(${answer:AnswerByGoal[Name->?N]}) )
   },

   gRule(R03):ForallRule -> ${
      \if (
         (?Q:QuestionByWS[DateOfBirth->?Y], DateOfBirth(?DoB))@WM
      )
      \then ( %deltaInsert(${answer:AnswerByGoal[DateOfBirth->?DoB]}) )
   },

   gRule(R04):ForallRule -> ${
      \if (
         (?Q:QuestionByWS[Gender->?Z], Gender(?G))@WM
      )
      \then ( %deltaInsert(${answer:AnswerByGoal[Gender->?G]}) )
   },

   gRule(R05):ForallRule -> ${
      \if (
         (?Q:QuestionByWS[
               CreditCardNo->?X,
               CreditCardHolder->?Y,
               CreditCardCVV->?Z])@WM,
               ( CreditCardNo(?CCN),
                 CreditCardHolder(?CCH),
                 CreditCardCVV(?CCCVV))@WM
      )
      \then (
         %deltaInsert(${answer:AnswerByGoal[
                        CreditCardNo->?CCN,
                        CreditCardHolder->?CCH,
                        CreditCardCVV->(?CCCVV)]})
      )
   }
].
```

Listing 4. Web Service semantic choreography specification for flight reservation

```
// Local ontology (will be stored on a separate file after deployed)
/*
flight(F100,Paris,Chicago,23,250).
flight(F101,Paris,Chicago,23,350).
flight(F102,Paris,Chicago,25,400).
flight(F103,Chicago,Paris,29,150).
flight(F104,Chicago,Paris,30,200).
flight(F105,Chicago,Paris,30,150).
*/
FlightReservationService:WebService.
FlightReservationService[
   importOntology -> ../Flight/WebServicesOntology.flr',
   capability -> ${
       pre -> ${ ?Req:RequestFlight[?X1->?Y1] },

       post -> ${ (?Res:Reservation[?X2->?Y2]) }
   },
   wsRule(R01):ForallRule -> ${
      \if (
          (?R:RequestFlight[From->?X,To->?Y,Departure->?Z,Return->?W])@WM,
         (flight(?fl_dep,?X,?Y,?Z,?priceDep))@WM,
         (flight(?fl_ret,?Y,?X,?W,?priceRet))@WM,
         (%sum(?priceDep,?priceRet,?priceTot))
      )
      \then (
         %deltaInsert(${tripChoice(?fl_dep,?fl_ret,?priceTot)}) ) },
   wsRule(R02):ForallRule -> ${
      \if
         (?T:Trip[Dep->?fl_dep,Ret->?fl_ret])@WM
      \then (
         %deltaInsert(${question:QuestionByWS[
                          Name->?X,
                          DateOfBirth->?Y,
                          Gender->?Z]})) },
   wsRule(R03):ForallRule -> ${
      \if
         (?A:AnswerByGoal[
                Name->?X,
                DateOfBirth->?Y,
                Gender->?Z])@WM
      \then (
         %deltaInsert(${question:QuestionByWS[
                          CreditCardNo->?XX,
                          CreditCardHolder->?YY,
                          CreditCardCVV->?ZZ]}) ) },
   wsRule(R04):ForallRule -> ${
      \if
         (?A:AnswerByGoal[
               CreditCardNo->?X,
               CreditCardHolder->?Y,
               CreditCardCVV->?Z])@WM
      \then (
         %deltaInsert(${validation:CreditCardValidation[
                          Number->?X,
                          Holder->?Y,
                          CVV->?Z]}) ) },
   wsRule(R05):ForallRule -> ${
      \if (
          (YesNoAnswer('Yes'))@WM,
          (trip:Trip[Dep->?fl_dep,Ret->?fl_ret])@WM )
      \then (
         %deltaInsert(${reservation:Reservation[
                          Number->11100,
                          Flight1->?fl_dep,
                          Flight2->?fl_ret]}) ) },
   wsRule(Bank_R01):ForallRule -> ${
      \if (
         (?R:CreditCardValidation[Number->?X,Holder->?Y,CVV->?Z])@WM,
          (DB_CreditCard(?X,?Y,?Z))@WM
      )
      \then (
         %deltaInsert(${YesNoAnswer('Yes')}) ) }
].
```

Rule `wsRule(R02)` in the web service specification poses a question asking for passenger name, date of birth and gender in the form of a frame. In the goal specification, the answer to each of these items are provided by rules `gRule(R01)`, `gRule(R02)`, and `gRule(R03)`. However, the left-hand side of rule `wsRule(R03)` in the web service choreography specification still can be proven when all items are inserted by the goal and the rule can fire.

3 Choosing a branch by the goal or web service

There are situations where either the goal or web service wants to continue the choreography in the specific branch they choose. For example, in an online payment scenario a goal can either choose to do the payment via credit card or PayPal. Depending on the goal choice, the choreography moves on. Here, we show this possibility in an abstract example. Listings 5 and 6 depict the semantic goal and web service choreography specifications.

Listing 5. Goal semantic choreography specification

```
// Local ontology (will be stored on a separate file after deployed)
/*
CreditCardNo('1234432156788765').
CreditCardHolder('PETER JACKSON').
CreditCardCVV(123).
PayPalThreshold(1000).
PayPal(UserName,'PetJack').
PayPal(Password,'1234').
*/
myGoal:Goal.
myGoal[
   importOntology -> '../PayPal/GoalsOntology.flr',

   capability -> ${
       pre -> ${ myRequest:RequestPurchase[Item->'ABC'] },
       post -> ${ ?R:PurchaseReceipt[?X->?Y] }
   },
   gRule(R01):ForallRule -> ${
      \if (
           (?Q:QuestionByWS[
                 price-> ?P,
                 paymentMethod->?PM])@WM,
                 (PayPalThreshold(?T))@WM , (?P > ?T)
      )
      \then (
        %deltaInsert(${answer:AnswerByGoal[
                           paymentMethod->'CreditCard']}) ) },
    gRule(R02):ForallRule -> ${
      \if (
           (?Q:QuestionByWS[
                 price-> ?P,
                 paymentMethod->?PM])@WM,
                 (PayPalThreshold(?T))@WM, (?P =< ?T)
      )
      \then (
        %deltaInsert(${answer:AnswerByGoal[
                           paymentMethod->'PayPal']}) ) },
   gRule(R03):ForallRule -> ${
      \if (
           (?Q:QuestionByWS[
                 CreditCardNo->?X,
                 CreditCardHolder->?Y,
                 CreditCardCVV->?Z])@WM,
                 (CreditCardNo(?CCN),
                     CreditCardHolder(?CCH),
                     CreditCardCVV(?CCCVV))@WM
      )
      \then (
         %deltaInsert(${answer:AnswerByGoal[
                           CreditCardNo->?CCN,
                           CreditCardHolder->?CCH,
                           CreditCardCVV->(?CCCVV)]}) ) },
   gRule(R04):ForallRule -> ${
      \if (
           (?Q:QuestionByWS[
                PayPalUserName->?UN,
                PayPalPassword->?PW])@WM,
                (PayPal(UserName,?myUN),PayPal(Password,?myPW))@WM
      )
      \then (
         %deltaInsert(${answer:AnswerByGoal[
                           PayPalUserName->?myUN,
                           PayPalPassword->?myPW]}) ) }
].
```

Listing 6. Web Service semantic choreography specification

```
// Local ontology (will be stored on a separate file after deployed)
/*
*/
OnlinePayment:WebService.
OnlinePayment[
   importOntology -> '../PayPal/WebServicesOntology.flr',

   capability -> ${
       pre -> ${ ?Req:RequestPurchase[?X1->?Y1] },
       post -> ${ ?Rec:PurchaseReceipt[?X2->?Y2] }
   },
   wsRule(R01):ForallRule -> ${
      \if (
         (?R:RequestPurchase[Item->?X])@WM
      )
      \then (
         %deltaInsert(${Q:QuestionByWS[
                            price-> 1700,
                            paymentMethod->_]}) ) },
   wsRule(R02):ForallRule -> ${
      \if
         (?A:AnswerByGoal[paymentMethod->'PayPal'])@WM
      \then (
         %deltaInsert(${Q:QuestionByWS[
                            PayPalUserName->_,
                            PayPalPassword->_]}) ) },
   wsRule(R03):ForallRule -> ${
      \if
         (?A:AnswerByGoal[paymentMethod->'CreditCard'])@WM
      \then (
         %deltaInsert(${Q:QuestionByWS[
                            CreditCardNo->_,
                            CreditCardHolder->_,
                            CreditCardCVV->_]}) ) },
   wsRule(R04):ForallRule -> ${
      \if
         (?A:AnswerByGoal[
               CreditCardNo->?X,
               CreditCardHolder->?Y,
               CreditCardCVV->?Z])@WM
      \then (
         %deltaInsert(${Rec1:PurchaseReceipt[Method->'CreditCard']}) ) },
   wsRule(R05):ForallRule -> ${
      \if
         (?A:AnswerByGoal[
               PayPalUserName->?myUN,
               PayPalPassword->?myPW])@WM
      \then (
         %deltaInsert(${Rec2:PurchaseReceipt[Method->'PayPal']}) ) }
].
```

In this example, the goal specification contains two rules which have complementary left-hand side conditions: `gRule(R01)` checks if the price of the requested item is more than the specified threshold (in this case 1000), and `gRule(R02)` checks if the price is less than or equal to the threshold. Based on the price, one of the answers `paymentMethod->'CreditCard'` or `paymentMethod->'PayPal'` is provided by the goal and the web service continues the choreography based on it. As it can be seen, the Flora-2 relational operators such as > or =< are available for the choreography specification.

4 Using utility predicates

A web service can offer some utility predicates to its requesters, so requesters can utilize these predicates in their choreography specifications. The signature of the predicate should be known to the requester at the time of writing specification. These predicates can be imported into the common ontology with static access; so both the web service and goal can utilize them. Here, we show the Flight Reservation System example utilizing the utility predicate `%cheapestFlight` to select the cheapest roundtrip flight among the suggested ones. The predicate is defined as below in the common ontology. It uses Flora-2 *min* operator to find the minimum value among the values which are unified with `?priceTot`.

```
%cheapestFlight(?price) :- ?price = min {?priceTot | tripChoice(?fl_dep,?fl_ret,?priceTot)@WM}.
```

Listings 7 and 8 contain the goal and web service choreography specifications respectively. To fit the listings on this page, local ontologies have been removed. The predicate `%cheapestFlight` is used in `gRule(R01)`.

Listing 7. Goal semantic choreography specification

```
myGoal:Goal.
myGoal[
   importOntology -> '../Flight/GoalsOntology.flr',
   capability -> ${
       pre -> ${
          myRequest:RequestFlight[
              From->'Paris',
              To->'Chicago',
              Departure->23,
              Return->30] },

       post -> ${ ?R:Reservation[?X->?Y] }
   },
   gRule(R01):ForallRule -> ${
      \if
         ( %cheapestFlight(?P),(tripChoice(?fl_dep,?fl_ret,?P))@WM )
      \then (
         %deltaInsert(${ trip:Trip[Dep->?fl_dep,Ret->?fl_ret]}) ) },
   gRule(R02):ForallRule -> ${
      \if (
           (?Q:QuestionByWS[
                 Name->?X,
                 DateOfBirth->?Y,
                 Gender->?Z])@WM,
                (Name(?N),DateOfBirth(?DoB),Gender(?G))@WM
      )
      \then (
         %deltaInsert(${answer:AnswerByGoal[
                            Name->?N,
                            DateOfBirth->?DoB,
                            Gender->?G]}) ) },
   gRule(R03):ForallRule -> ${
      \if (
           (?Q:QuestionByWS[
                 CreditCardNo->?X,
                 CreditCardHolder->?Y,
                 CreditCardCVV->?Z])@WM,
                (CreditCardNo(?CCN),
                      CreditCardHolder(?CCH),
                      CreditCardCVV(?CCCVV))@WM
      )
      \then (
         %deltaInsert(${answer:AnswerByGoal[
                            CreditCardNo->?CCN,
                            CreditCardHolder->?CCH,
                            CreditCardCVV->(?CCCVV)]}) ) }
].
```

Listing 8. Web Service semantic choreography specification

```
FlightReservationService:WebService.
FlightReservationService[
   importOntology -> '../Flight/WebServicesOntology.flr',
   capability -> ${
       pre -> ${ ?Req:RequestFlight[?X1->?Y1] },
       post -> ${ (?Res:Reservation[?X2->?Y2]) }
   },
   wsRule(R01):ForallRule -> ${
      \if (
         (?R:RequestFlight[From->?X,To->?Y,Departure->?Z,Return->?W])@WM,
          (flight(?fl_dep,?X,?Y,?Z,?priceDep))@WM,
          (flight(?fl_ret,?Y,?X,?W,?priceRet))@WM,
          (%sum(?priceDep,?priceRet,?priceTot))
      )
      \then (
         %deltaInsert(${tripChoice(?fl_dep,?fl_ret,?priceTot)}) ) },
   wsRule(R02):ForallRule -> ${
      \if
         (?T:Trip[Dep->?fl_dep,Ret->?fl_ret])@WM
      \then (
         %deltaInsert(${question:QuestionByWS[
                            Name->?X,
                            DateOfBirth->?Y,
                            Gender->?Z]}) ) },
   wsRule(R03):ForallRule -> ${
      \if
         (?A:AnswerByGoal[
               Name->?X,
               DateOfBirth->?Y,
               Gender->?Z])@WM
      \then (
         %deltaInsert(${question:QuestionByWS[
                            CreditCardNo->?XX,
                            CreditCardHolder->?YY,
                            CreditCardCVV->?ZZ]}) ) },
   wsRule(R04):ForallRule -> ${
      \if
         (?A:AnswerByGoal[
               CreditCardNo->?X,
               CreditCardHolder->?Y,
               CreditCardCVV->?Z])@WM
      \then (
         %deltaInsert(${validation:CreditCardValidation[
                            Number->?X,Holder->?Y,CVV->?Z]})
      )
   },
   wsRule(R05):ForallRule -> ${
      \if (
           (YesNoAnswer('Yes'))@WM,
            (trip:Trip[Dep->?fl_dep,Ret->?fl_ret])@WM
      )
      \then (
         %deltaInsert(${reservation:Reservation[
                            Number->11100,
                            Flight1->?fl_dep,
                            Flight2->?fl_ret]}) ) },
   wsRule(Bank_R01):ForallRule -> ${
      \if (
           (?R:CreditCardValidation[
                  Number->?X,
                  Holder->?Y,
                  CVV->?Z])@WM,
          (DB_CreditCard(?X,?Y,?Z))@WM
      )
      \then (
         %deltaInsert(${YesNoAnswer('Yes')})
      )
   }
].
```

5 Shipwire example

Shipwire is "a Fortune 100 company helping brands expand to new markets all around the world with [its] technology platform and distribution centers in more than 45 countries" [1]. The company has authenticated open API facilities for developers. As a tutorial of working with APIs, Shipwire provides a scenario consisting of step-by-step REST requests and responses showing how a developer can accomplish a shipment order. This scenario is semi-choreographic (because it involves human specific behavior as well) and is described in

Figure 9 [2].

1. *The app checks stock availability of products and updates its catalog.*
2. *Audrey browses the catalog and picks items to fill her shopping cart.*
3. *When Audrey is ready, she proceeds to checkout, where she must select among different shipment rates and services.*
4. *After Audrey confirms checkout, the app places a shipping order with Shipwire.*
5. *Audrey realizes her order is one item short, so she modifies the original order and checks out again. The app updates the order with Shipwire.*
6. *Once Shipwire ships Audrey's packages, the app automatically sends Audrey a shipping confirmation email with her tracking number.*

Figure 9. Shipwire semi-choreographic scenario [2]

Here, we show how such choreography scenario can also be modeled by our choreography specification (Listings 9 and 10). To make the presentation clearer, we removed un-essential details; however, the original scenario can be modeled in full in the same way.

Listing 9. Goal choreography specification for user

```
// Local ontology (will be stored on a separate file after deployed)
/*
myUserName('Audrey').
myAuth('TG9vayBhdCB0aGF0OyBEdWNrcy4uLm9uIGEgbGFrZSEK').
myAddress('6501 Railroad Avenue SE-Room 315').
WantToBuy('Laura-s_Lament',10).
*/
myGoal:Goal.
myGoal[
   importOntology -> '../Shipwire/GoalsOntology.flr',

   capability -> ${
       pre -> ${
             run:System[state->on]
       },
       post -> ${
             RESPONSE:MESSAGE[
                 PurchaseDone->?_X,
                 TrackingNumber->?_Y
             ]
       }
   }
   ,
   gRule(R01):ForallRule -> ${
      \if (
        (
             (\+ g_Rule:Control[R01->off])@WM,
             (run:System[state->on])@WM,
             myUserName(?user)@WM,
             myAuth(?auth)@WM,
             WantToBuy(?sku,?_quan)@WM
        )
      )
      \then (
         %deltaInsert(${g_Rule:Control[R01->off]}),
         %deltaInsert(${REQUEST:MESSAGE[
                     TYPE->'Search',
                     SKU->?sku,
                     USER->?user,
                     AUTH->?auth
                 ]})
      )
   }
   ,
   gRule(R02):ForallRule -> ${
      \if (
        (
             (\+ g_Rule:Control[R02->off])@WM,
             (RESPONSE:MESSAGE[
                 Available->'TRUE',
                 ProductID->?PrdId
             ])@WM,
             myAuth(?auth)@WM,
             WantToBuy(?_prod,?quan)@WM
        )
      )
      \then (
         %deltaInsert(${g_Rule:Control[R02->off]}),
         %deltaInsert(${REQUEST:MESSAGE[
                             TYPE->'Order',
                             ORDER->?PrdId,
                             QUANTITY->?quan,
                             AUTH->?auth
```

Listing 10. Web Service choreography specification of simlulated Shipwire

```
// Local ontology (will be stored on a separate file after deployed)
/*
Product('Laura-s_Lament','1361533',20).
User('Audrey', 'TG9vayBhdCB0aGF0OyBEdWNrcy4uLm9uIGEgbGFrZSEK').
*/

Shipwire:WebService.
Shipwire[
   importOntology -> ../Shipwire/WebServicesOntology.flr',

   capability -> ${
       pre -> ${
          REQUEST:MESSAGE[?_X1->?_Y1]
       },

       post -> ${
          RESPONSE:MESSAGE[?_X2->?_Y2]
       }
   }
   ,
   wsRule(R01):ForallRule -> ${
      \if (
            (\+ ws_Rule:Control[R01->off])@WM,
            REQUEST:MESSAGE[
                TYPE->'Search',
                SKU->?sku,
                USER->?user,
                AUTH->?auth
            ]@WM,
            User(?user, ?auth)@WM,
            Product(?sku,?PrdId,?_Quan)@WM
      )
      \then (
        %deltaInsert(${ws_Rule:Control[R01->off]}),
        %deltaInsert(${RESPONSE:MESSAGE[
                            Available->'TRUE',
                            ProductID->?PrdId
                        ]})
      )
   }
   ,
   wsRule(R02):ForallRule -> ${
      \if (
            (\+ ws_Rule:Control[R02->off])@WM,
            REQUEST:MESSAGE[
                TYPE->'Order',
                ORDER->?PrdId,
                QUANTITY->?quanReq,
                AUTH->?auth
            ]@WM,
            Product(?_sku,?PrdId,?Quan)@WM,
            ?Quan >= ?quanReq
      )
      \then (
        %deltaInsert(${ws_Rule:Control[R02->off]}),
        %deltaInsert(${RESPONSE:MESSAGE[
                            AcceptOrder->'TRUE',
                            ProductID->?PrdId
                        ]})
      )
   }
   ,
   wsRule(R03):ForallRule -> ${
```

[1] https://www.shipwire.com/
[2] https://www.shipwire.com/w/developers/tutorial/

```
                        ]})
      )
   }
   ,
   gRule(R03):ForallRule -> ${
      \if (
         (
            (\+ g_Rule:Control[R03->off])@WM,
            (RESPONSE:MESSAGE[
                 AcceptOrder->'TRUE',
                 ProductID->?PrdId
             ])@WM,
             myAuth(?auth)@WM,
             WantToBuy(?_prod,?quan)@WM
         )
      )
      \then (
         %deltaInsert(${g_Rule:Control[R03->off]}),
         %deltaInsert(${REQUEST:MESSAGE[
                            TYPE->'Checkout',
                            ORDER->?PrdId,
                            QUANTITY->?quan,
                            AUTH->?auth
         ]})
      )
   }
   ,
   gRule(R04):ForallRule -> ${
      \if (
         (
            (\+ g_Rule:Control[R04->off])@WM,
            (RESPONSE:MESSAGE[
                    Question->'ADDRESS'
             ])@WM,
             myAddress(?add)@WM
         )
      )
      \then (
         %deltaInsert(${g_Rule:Control[R04->off]}),
         %deltaInsert(${REQUEST:MESSAGE[
                               TYPE->'Answer',
                               Data->?add
         ]})
      )
   }

].
```

```
      \if (
            (\+ ws_Rule:Control[R03->off])@WM,
            REQUEST:MESSAGE[
               TYPE->'Checkout',
               ORDER->?PrdId,
               QUANTITY->10,
               AUTH->'TG9vayBhdCB0aGF0OyBEdWNrcy4uLm9uIGEgbGFrZSEK'
            ]@WM
      )
      \then (
        %deltaInsert(${ws_Rule:Control[R03->off]}),
         %deltaInsert(${RESPONSE:MESSAGE[
                              Question->'ADDRESS'
                           ]})
      )
   }
   ,
   wsRule(R04):ForallRule -> ${
      \if (
            (\+ ws_Rule:Control[R04->off])@WM,
            REQUEST:MESSAGE[
               TYPE->'Answer',
               Data->?_address
            ]@WM
      )
      \then (
        %deltaInsert(${ws_Rule:Control[R04->off]}),
         %deltaInsert(${RESPONSE:MESSAGE[
                              PurchaseDone->'TRUE',
                              TrackingNumber->'1234567890'
                           ]})
      )
   }
].
```

In this example, each rule can only be fired once. We enforce this restriction by putting proper flags in the rules' left-sides. If a rule gets fired, subsequent firing of the same rule is prevented since the flag will be false after the first firing.

Step 5 of Shipware scenario denotes the situation where the user decides to change her/his mind. Although such decisions are normally made by a human agent, they can also be modeled by our specifications. For example, a predicate like `UserChangeMind('TRUE')` can be inserted as a pre-condition at the beginning and by using the above mentioned flagging technique, we can give only one chance to the goal to update its choice. Another way is to define a random function that activates a proper condition, simulating the user's change of mind.

6 Using well-known vocabularies (schema.org)

Flora-2 inherently supports defining the membership relation. The frame `A:C[B->1]` means that object `A` is a member of class `C` and its `B` attribute has value `1`. In general, membership relation is defined by the `:` operator. This feature is very suitable to link Flora-2 concepts to well-known vocabularies like the one available on `schema.org`[3]. For example, we can define the `flight` predicate used in the previous examples as an instance of `http://schema.org/Flight` by `flight:'http://schema.org/Flight'`. This relation can be stored in the common ontology and can be validated during the choreography runs. Here we rewrite the authentication example (Example 1) with `schema.org` annotations. Table 2 shows the concepts which can be mapped to the current version of `schema.org` vocabulary.

Table 2. Schema.org mapping

Concept	schema.org type
QuestionByWS	http://schema.org/Question
AnswerByGoal	http://schema.org/Answer (stack overflow answer)
myUserName	http://schema.org/identifier
myPassword	http://schema.org/accessCode
UserName	http://schema.org/identifier
Password	http://schema.org/accessCode
Message	http://schema.org/Message
Goal	http://schema.org/agent
WebService	http://schema.org/Service

Listings 11 and 12 show the authentication goal and web service specifications respectively using the `schema.org` vocabulary. Vocabulary validation lines are bolded.

Listing 11. Goal choreography specification for semantic authentication

```
// Local ontology (will be stored on a separate file after deployment)
/*
myUserName('Riccardo').
myPassword('98765').
*/
myGoal:Goal.
myGoal[
   importOntology -> '../Auth/GoalsOntology.flr',

   capability -> ${
       pre -> ${
          ar:AuthenticationRequest[UserName->?UN, Password->?PW],
          AuthenticationRequest(UserName,?UN),
          AuthenticationRequest(Password,?PW)
       },

       post -> ${
          ?AV:AuthenticationValidation[?X->?Y] \or
          AuthenticationValidation() \or
          AuthenticationValidation(?Z) \or
          AuthenticationValidation(?W,?S)
       }
   },
   gRule(R01):ForallRule -> ${
      \if (
         ?X:QuestionByWS[UserName->?Y]@WM,
         myUserName(?UN)@WM,
         QuestionByWS:'http://schema.org/Question',
         myUserName:'http://schema.org/identifier'
      )
      \then (
        %deltaInsert(${A:AnswerByGoal[UserName->?UN]})
      )
   },
   gRule(R02):ForallRule -> ${
      \if (
         ?X:QuestionByWS[Password->?Y]@WM,
         myPassword(?PW)@WM,
         QuestionByWS:'http://schema.org/Question',
         myPassword:'http://schema.org/accessCode'
      )
      \then (
        %deltaInsert(${A:AnswerByGoal[Password->?PW]})
      )
   },
   gRule(R03):ForallRule -> ${
      \if (
         QuestionByWS(UserName,?X)@WM,
         myUserName(?UN)@WM,
         QuestionByWS:'http://schema.org/Question',
         myUserName:'http://schema.org/identifier'
      )
      \then (
        %deltaInsert(${AnswerByGoal(UserName,?UN)})
      )
   },
   gRule(R04):ForallRule -> ${
      \if (
         QuestionByWS(Password,?X)@WM,
         myPassword(?PW)@WM,
         QuestionByWS:'http://schema.org/Question',
         myUserName:'http://schema.org/identifier'
      )
      \then (
        %deltaInsert(${AnswerByGoal(Password,?PW)})
      )
   }].
```

Listing 12. Web Service choreography specification of semantic authentication

```
// Local ontology (will be stored on a separate file after deployment)
/*
DB_UserName_Password('PeterJM','12345').
DB_UserName_Password('Angel83','112233').
DB_UserName_Password('Riccardo','98765').
DB_UserName_Password('Agent007','7777777').
*/
AuthenticationService:WebService.
AuthenticationService[
   importOntology -> '../Auth/WebServicesOntology.flr',

   capability -> ${
       pre -> ${
          ?AR:AuthenticationRequest[?X->?Y]
       },

       post -> ${
          ?AV:AuthenticationValidation[?X->?Y]
       }
   },
   wsRule(R01):ForallRule -> ${
      \if (
         (?AR:AuthenticationRequest[?X->?Y])@WM
      )
      \then (
         %deltaInsert(${Q:QuestionByWS[UserName->_]}),
         %deltaInsert(${Q:QuestionByWS[Password->_]})
      )
   },
   wsRule(R02):ForallRule -> ${
      \if (
         (?X:AnswerByGoal[UserName->?UN])@WM,
         (?Y:AnswerByGoal[Password->?PW])@WM,
         DB_UserName_Password(?UN,?PW)@WM,
         AnswerByGoal:'http://schema.org/Answer',
         UserName:'http://schema.org/identifier',
         Password:'http://schema.org/accessCode'
      )
      \then (
         %deltaInsert(${av:AuthenticationValidation[
                                          UserName->?UN,
                                          Password->?PW]})
      )
   },
   wsRule(R03):ForallRule -> ${
      \if (
         (?X:AnswerByGoal[UserName->?UN])@WM,
         (?Y:AnswerByGoal[Password->?PW])@WM,
         (\+ DB_UserName_Password(?UN,?PW))@WM,
         AnswerByGoal:'http://schema.org/Answer',
         UserName:'http://schema.org/identifier',
         Password:'http://schema.org/accessCode'
      )
      \then (
         %deltaInsert(${M:Message[
                         WS->'Incorrect username or password.']})
      )
   }
].
```

[3] http://schema.org/

7 HTTP messages

As mentioned before REST APIs are very popular. Here, we show how HTTP messages can be encoded to Flora-2 choreography specifications. In Example 5, we demonstrated a choreographic scenario provided by the Shipwire website. We abstractly showed how this scenario can be modeled with the choreography specification. At a lower level, REST APIs are called in single step request/response pairs and because HTTP is a stateless protocol, control flow is managed at higher levels.

Here, we show how a sample REST request/response pair can be modeled by the Flora-2 choreography specification. The sample GET request/response taken from TIBCO® [4] and is shown below:

Request	`http://localhost:8090/tpmRest/v1/participants/transports/all?participantName=partner1&protocolName=EZComm`
Response	`{"result":[{"name":"file","type":"FILE"},{"name":"http","type":"HTTP"}]}`

We use the JSON to Flora-2 conversion presented in Appendix E to specify the JSON response provided by the web service. Listings 13 and 14 show the goal and web service specifications. We use the attribute ID to make correlation between the request and response messages.

Listing 13. Goal choreography specification for semantic GET request

```
myGoal:Goal.
myGoal[
  importOntology -> '../Tibco/GoalsOntology.flr',

  capability -> ${
    pre -> ${ run:System[state->on] },
    post -> ${
        RESPONSE:'HTTP_1.1_MESSAGE'[
          ID->123,
          STATUS->200,
          BODY->?format[content -> ?top(?topId)]
        ]
    }
  }
  ,
  gRule(R01):ForallRule -> ${
    \if (
    (
      (\+ g_Rule:Control[R01->off])@WM,
      (run:System[state->on])@WM
    )
    )
    \then (
      %deltaInsert(${g_Rule:Control[R01->off]}),
      %deltaInsert(${
         REQUEST:'HTTP_1.1_MESSAGE'[
             ID->123,
             METHOD->'GET',
             URI->'http://localhost:8090/tpmRest/v1/
                                  participants/transports/all',
                PARAMS->participantName('partner1'),
                PARAMS->protocolName('EZComm')
                // , HEADER->Accept('text/json'),
                // BODY->Optional
            ]})
    )
  }
].
```

Listing 14. Web Service choreography specification of semantic GET response

```
Tibco:WebService.
Tibco[
  importOntology -> '../Tibco/WebServicesOntology.flr',

  capability -> ${
    pre -> ${ REQUEST:'HTTP_1.1_MESSAGE'[?_X1->?_Y1] },

    post -> ${ RESPONSE:'HTTP_1.1_MESSAGE'[?_X2->?_Y2] }
  }
  ,
  wsRule(R01):ForallRule -> ${
    \if (
      (\+ ws_Rule:Control[R01->off])@WM,
      REQUEST:'HTTP_1.1_MESSAGE'[
            ID->?id,
            METHOD->'GET',
            URI->?uri,
            PARAMS->?_param(?val)
            // , HEADER->Optional,
            // BODY->Optional
      ]@WM
      // , call the appropriate function with
      // the provided parameters
    )
    \then (
      %deltaInsert(${ws_Rule:Control[R01->off]}),
      %deltaInsert(${RESPONSE:'HTTP_1.1_MESSAGE'[
              ID->?id,
              STATUS->200,
              BODY->json[content->object(obj_01)],

              object(obj_01,'result')-> array(arr_01),

              array(arr_01,1)->object(obj_02),

              object(obj_02,'name')->'file',
              object(obj_02,'type')->'FILE',

              array(arr_01,2)->object(obj_03),

              object(obj_03,'name')->'http',
              object(obj_03,'type')->'HTTP'
            ]})
    )
  }
].
```

Since there is only one request/response step, the request method can be also defined in *goal.pre*. In this form, goal will be rule-free however.

[4] https://docs.tibco.com/pub/tpm-rest/1.0.1/doc/html/GUID-BAA2DC07-D7DC-49BD-80A5-B4998B56B9BF.html

APPENDIX D: CHOREOGRAPHY ENGINE PREDICATE LIST

Predicate	Defined in	Functionality
%checkAllRHS(?gOrWS,?List)	Library.flr	Picks each of the collected RHSs in the ?List and passes it to %checkRHS. It is used to see if there is any usage of membership operators in deltaDelete actions.
%checkExistanceOfColon (?gOrWS,?Action)	Library.flr	If the specified action (?Action) is deltaDelete, then it checks whether there is any membership operator used in the action subject.
%checkMembershipOperator(?Obj)	Library.flr	Converts the reified Flora-2 object (?Obj) into string and then passes its characters one-by-one to %noColonExists.
%checkModule(?M)	Utility.flr	Checks existence of module ?M, if not creates it.
%checkRHS(?gOrWS,?P)	Library.flr	If ?P is a pair it decomposes it, otherwise it sends ?P to %checkExistanceOfColon.
%checkUsageOfMembershipOperatorInDeltaDelete (?gOrWS)	Library.flr	Prepares the list of all RHSs belonging to ?gOrWs which can be either a goal or a web service.
%contained(?X, ?A)	Utility.flr	Checks whether ?X exists in ?A. ?A is conjunction of literals.
%contradictory(?WM, ?DeltaWM)	Choreography.flr	Checks whether there are any contradictory actions who are pending in Delta Working Memory. WM and DeltaWM are in ?WM and ?DeltaWM respectively.
%convertReifiedObjectModule (?obj1, ?M1, ?M2, ?obj2)	Library.flr	Copies the reified object ?obj1 into object ?obj2 with the module name changed from ?M1 to ?M2.
%copyReifiedObjectIntoModule (?obj1, ?M1, ?M2, ?obj2)	Library.flr	Makes a copy of the reified object ?obj1 in module ?M1 into ?obj2 and inserts it in ?M2.
%debug(?X)	Utility.flr	Toggles the debug mode.
%deltaDelete(?obj)	Library.flr	Inserts deletion of an object into delta working memory.
%deltaInsert(?obj)	Library.flr	Inserts insertion of an object into delta working memory
%deltaMakesAChange (?WM, ?DeltaWM)	Choreography.flr	Checks whether the pending actions in delta working memory make a new state for choreography
%deltaUpdate(?objOld, ?objNew)	Library.flr	Inserts updating of an object into delta working memory
%eraseModule(?M)	Utility.flr	Deletes the content of the specified module.
%extractConcepts (?Str, ?LstIn, ?LstOut)	Library.flr	Finds and extracts the concept terms in the string ?Str.
%extractPredicates (?Str, ?LstIn, ?LstOut)	Library.flr	Finds and extracts the terms which could be object names or predicate names in the string ?Str
%filterOutPredicates (?List, ?LstIn, ?LstOut)	Library.flr	Among list of object names and predicate names represented by ?List ,it extracts the predicate names.
%giveElementAt(?L,?n,?elementAt)	Utility.flr	Returns the ?nth element of the list ?L in ?elementAt.
%importOntology(?X,?module)	Library.flr	Loads the specified ontology into the specified module.
%initializations	Utility.flr	Initializes the choreography engine parameters (currently initialize the RNG seed).
%insertGoalPre (?goal,?M)	Library.flr	Inserts the precondition of goal ?goal into the ?M module.
%invoke(WEBSERVICE,?X)	Choreography.flr	Fires the forall rules of a web service or a goal.
%invokeChoose(WEBSERVICE,?X)	Choreography.flr	Fires the choose rules of a web service or a goal.
%isNotFrame([?H|?T])	Library.flr	Determines if a string represents a Flora-2 frame.
%makeFileAddress (?dir,?name,?fileAddress)	Utility.flr	Concatenates the directory path and the file name to make a complete file address.
%mergeDeltaIntoWM	Choreography.flr	Does actual actions which were pending in delta working memory and makes a new state of working memory.
%noColonExists([])	Library.flr	Checks whether there is a ':' in the passed string.
%pause()	Utility.flr	Stops the reasoner to let the user to enter something.
%prepareModule(?module)	Utility.flr	Creates a new module if it does not exist.
%preProcessCheckings(?goal,?WS)	Library.flr	Checks the access modes in the goal precondition and the transition rules.
%prove(?X)	Library.flr	Tries to prove any logical expression.
%proveGoalPost(?goal)	Library.flr	Tries to prove the goal post condition.
%rand(?L,?U,?R)	Utility.flr	Generates a random integer number in the specified range.
%readTheTerm (?Str, ?termIn, ?termOut, ?remainder)	Utility.flr	Extracts terms after specific characters in the string ?Str one at a time.
%reformatToString (?In, ?Str)	Utility.flr	Converts a Flora-2 object definition into a string.
%removeParenthesis (?Str, ?In, ?Out)	Utility.flr	Scans a string and reads the characters until reaching an opening parenthesis.
%replaceAll (?Str,?Pattern,?Replace,?NewStr)	Utility.flr	Replaces all the substrings matching ?Pattern with the string ?Replace and returns the new string by ?NewStr.

%runChoreography(?goal, ?WS)	Choreography.flr	The main predicate of choreography execution.
%runGoalRules(?goal)	Choreography.flr	Runs the transition rules of the goal.
%runWsRules(?WS)	Choreography.flr	Runs the transition rules of the web service.
%showModule(?M)	Utility.flr	Shows the content of a module in the output.
%start(?goal,?WS)	Choreography.flr	The predicate which should be called to begin the choreography execution engine.
%startsWith (?Str,?Pattern,?Rest)	Utility.flr	Checks whether the string ?Str starts with the string ?Pattern.
%watch(?X)	Utility.flr	Prints the parameter represented by ?X in the output.
%watchln()	Utility.flr	Prints the parameter represented by ?X and a new line character in the output.
%check(?gOrWs,?X)	ModeChecking.flr	Checks if the LHS and RHS of a rule (belonging to goal or web service) are using correct access modes.
%checkAll(?gOrWs,?L)	ModeChecking.flr	Extracts the rules of goal and web service in the list ?L and passes them to %check one-by-one.
%checkAllFramesModes (?gOrWS, ?reOrWr,?L)	ModeChecking.flr	Extracts the concept names from the list ?L and passes them to %checkFrameMode one-by-one.
%checkAllPredicatesModes (?gOrWS, ?reOrWr,?L)	ModeChecking.flr	Extracts the predicate names from the list ?L and passes them to %checkPredicateMode one-by-one.
%checkFrameMode(GOAL,READ,?F)	ModeChecking.flr	Checks the access modes of the specified frame based on the reading/writing of goal/web service
%checkGoalPreMode(?fOrP,?F)	ModeChecking.flr	Checks the access modes of goal precondition.
%checkModeDeltaDelete (GOAL/WS,READ/WRITE,?N)	ModeChecking.flr	Checks the access modes of the either frames or predicates used in deltaDelete.
%checkModeOfDeltaDeleteObject (?gOrWs, ?Z)	ModeChecking.flr	Separates deltaDelete actions from the others in RHS and passes them to %checkModeDeltaDelete one-by-one.
%checkModeOfDeltaDeleteObjects (?gOrWs, ?Z)	ModeChecking.flr	If RHS (?Z) contains more than one actions, it decomposes it; otherwise passes it to %checModeDeltaDelete.
%checkPredicateMode (GOA/WS,READ/WRITE,?F)	ModeChecking.flr	Checks the access modes of the specified predicate based on the reading/writing of goal/web service
%decomposeRHS (?X, ?LstIn, ?LstOut)	ModeChecking.flr	Makes a list containing all the reified objects and predicates existing in delta actions.
%extractConceptsForDeltaDelete (?Str, ?LstIn, ?LstOut)	ModeChecking.flr	Finds the concepts of the object names in the string ?Str.
%preProcessCheckModes (?gOrWs,?X)	ModeChecking.flr	Checks the access modes in the LHS and RHS of the transition rules.
%preProcessCheckModesForGoalPre (?goal)	ModeChecking.flr	Checkes the access modes in the goal precondition.
%readTheTermForDeltaDelete (?Str, ?termIn, ?termOut, ?remainder)	ModeChecking.flr	Extracts a term in the string ?Str which exists in the deltaDelete object or predicate.

Appendix E: Converting JSON to Flora-2

Currently, REST (REpresentational State Transfer) is the most popular architecture in the development of web services. In RESTful architectures, message passing and data transfer is done via HTTP over the network. Clients use methods such as GET, POST, and PUT to trigger actions or to retrieve resources held by web services. Web services, on the other hand, respond to these methods with HTTP messages containing formatted (and possibly annotated) information. The dominant formats in use are XML (eXtensible Markup Language) and JSON (JavaScript Object Notation). Between these two, JSON is the preferred format in REST because of its simplicity and readability. However, as it is a minimal type-free data format it does not support semantic annotations. It should be kept in mind that XML and JSON are not equivalent because the former is a language with schema, types and links but the latter is just a data presentation format. JSON-LD (JSON for Linked-Data)[5] is a typed version of JSON and has the potential to be used in semantic based systems, but it is not integrated with the most of the popular available web services yet. In this appendix we present a scheme to map JSON context to an equivalent Flora-2 context. This scheme paves the way for our Flora-2 choreography solution to interact with the currently available REST APIs.

JSON supports only two types of structure, namely *objects* and *arrays*. Objects are anonymous and contain comma separated key-value pairs. Keys are always strings and values can be other objects, string/Boolean/numerical literals, or arrays. Arrays contain series of objects and can be nested as well. JSON grammar is given in
Figure 10.

object →	*array* →	*value* →
{ }	**[ ]**	*string*
{ *members* **}**	**[** *elements* **]**	*number*
members →	*elements* →	*object*
pair	*value*	*array*
pair **,** *members*	*value* **,** *elements*	**true**
pair →		**false**
string **:** *value*		**null**

Figure 10. JSON grammar [6]

In Flora-2, frames model objects. Unlike JSON, frames have names in the form of object identifiers and can be instantiated from a concept (class). Frames are composed of attribute-value pairs which are similar to JSON key-value pairs. But because objects are anonymous in JSON it is not straightforward to directly map key-value pairs in JSON to attribute-value pairs in Flora-2. Another consideration is that in JSON, objects are referenced by their relative position and path in the whole structure and there is no direct pointer to a specific data, but in Flora-2 frames are decomposed and stored in form of predicates. Data in Flora-2 is retrieved by unification, so accessing data is fundamentally different from JSON. Taking into account the above mentioned issues, below we present a mapping scheme between JSON and Flora-2.

The top-level entity in JSON can be either an object or an array. We can show this by frames `json[content->object(obj_id)]` or `json[content->array(arr_id)]`.

In JSON, key-value pairs belonging to the same object are wrapped together inside the object. We can show this binding by combining predicate and frame notations in Flora-2. For example, the attribute-value pairs `object(id_1,a)->1` and `object(id_1,b)->2` tell that both attributes `a` and `b` belong to the object with id `id_1` and their values are `1` and `2` respectively. In this way, the query `object(id_1,?X)->?Y` returns all attribute-value pairs belonging to the object with id `id_1`. Moreover, `id_1` can be used as a direct reference to the object; in contrast to JSON wherein objects are anonymous.

The elements of an array in JSON are referenced by their position in a comma separated list. To show JSON lists in Flora-2 we can use again predicate and frame notations. For example, the attribute-value pair `array(id_1, 2)->a` says that the second element in array `id_1` has the value `a`.

An array which itself is a value of an attribute can be represented in Flora-2 as the predicate `array` whose parameter is the (generated) id of the array. For example the JSON object `{"someArray": [10,20]}` can be represented in Flora-2 by the facts `object(id_1, "someArray")->array(id_2)` , `array(id_2, 1)->10`, and `array(id_2, 2)->20` where `id_1` and `id_2` are automatically generated Flora-2 object identifiers. Using the Flora-2 reasoner, one can query such composite structures in a simple manner. For example, a query such as `someFrame:someConcept[object(id_1, a)->array(id_2), array(id_2, ?X)->?Y]` returns all the elements' index-value pairs of array `id_2` which is in turn the value of attribute `a` of object `id_1`.

Figure 11 depicts an example of REST response message, both in JSON and its equivalent Flora-2 notations. It is obvious that data presented in Flora-2 can be simply queried and reasoned about, and these are not possible with JSON.

JSON	Flora-2
`[`	`json[content -> array(arr_01)].`
`  {`	
`    "firstName": "John",`	`array(arr_01,1) -> object(obj_01).`
`    "lastName" : "doe",`	`object(obj_01,"firstName") -> "John".`
`     "age"      : 26,`	`object(obj_01,"lastName") -> "doe".`
`     "address"  : {`	`object(obj_01,"age") -> 26.`
`       "streetAddress": "naist street",`	`object(obj_01,"address") -> object(obj_02).`
`       "city"         : "Nara",`	
`       "postalCode"   : "630-0192"`	`object(obj_02, "streetAddress") -> "naist street".`
`  },`	`object(obj_02, "city") -> "Nara".`

[5] https://json-ld.org/
[6] http://www.json.org/

```
      "phoneNumbers": [
        {
          "type": "iPhone",
          "number": "0123-4567-8888"
        },
        {
          "type"  : "home",
          "number": "0123-4567-8910"
        }
      ]
    ]
```

```
    object(obj_02, "postalCode") -> "630-0192".

    array(arr_01,2) -> object(obj_03).
    object(obj_03,"phoneNumbers") -> array(arr_02).
    array(arr_02,1) -> object(obj_04).
    object(obj_04,"type") -> "iPhone".
    object(obj_04,"number") -> "0123-4567-8888".

    array(arr_02,2) -> object(obj_05).
    object(obj_05,"type") -> "home".
    object(obj_05,"number") -> "0123-4567-8910".
```

Figure 11. An example of JSON to Flora-2 conversion