Modelling and Transforming the Behavioural aspects of Web Services

Behzad Bordbar and Athanasios Staikopoulos

School of Computer Science, University of Birmingham, Birmingham B15 2TT, UK
B.Bordbar@cs.bham.ac.uk, A.Staikopoulos@cs.bham.ac.uk

Abstract. This paper introduces the modelling, mapping and transformation of behavioural aspects of interacting Web services, within the context of Model Driven Architecture (MDA). There are certain systems, such as Web services, where the dynamic aspects are of high importance and need to be considered during the modelling and transformation process, in order to create accurate representations in their target domains. To demonstrate the approach, a realistic example is presented, involving a number of Web services, participating in a business process expressed as choreography of exchanged messages.

Introduction

Currently, the research on model transformations within Model Driven Architecture (MDA) is restricted among models, including only their structural characteristics, such as their content and their interrelations. In this paper we apply model mappings and transformations among ***behavioural models*** and express the way the various components collaborate, in order to implement different system functionalities and behaviours. Transforming the behaviour of a system as long as its structure results to completely defined new models.

In particularly, if one considers a number of Web services standards, such as the Business Process Execution Language (BPEL) [2] and the Web services Choreography Interface (WSCI) [17] that represent business processes and choreographies of message exchanges among Web service participants respectively, one then understands that representing and mapping their behaviours is of primary importance.

In order to illustrate the applicability and feasibility of the approach presented, a case study based on a realistic scenario is considered. Thus, specific models, metamodels, mappings and transformation rules have been introduced, for modelling, mapping and transforming a UML activity diagram accordingly to a scenario implemented with Web Services Choreography Interfaces.

Preliminaries

Web services [3][4] are a set of technologies allowing applications to communicate with each other across the Internet. Among the technologies used are the Extensible Markup Language (XML) [16], the Web Service Description Language (WSDL) [18] and the Web Services Choreography Interface (WSCI) [17]. XML can be used either as a meta-language or for structuring and interchanging data. WSDL provides a model in an XML format, for describing Web services as a set of exposed operations and WSCI as an XML based interface, for describing the message exchanges among various Web service participants and their choreography. WSCI works in conjunction with WSDL and describes the dynamic interfaces of the Web services.

The Model Driven Architecture [1][7] is an emerging technology for software development. MDA promotes the automatic creation of models and code by a series of transformations. Initially, models are designed with a high level of abstraction that are independent of any implementation technology and called Platform Independent Models (PIMs). Then a PIM may be transformed to a Platform Specific Model (PSM) that is tailored to a specific implementation technology and finally to the actual code.

In MDA all models are based on a specific metamodel that defines the language that the model is created in. Finally, all MDA metamodels are based on a common metamodel called Meta Object Facility (MOF) [12]. Model transformations [6][11] are defined by transformation rules in a transformation language like OCL [13], ATLAS [5] etc and they are executed by specialised tools. According to a QVT RFP [14] the transformations are defined at metamodel level (M2) and executed at model level (M1) as illustrated on the Figure 1.

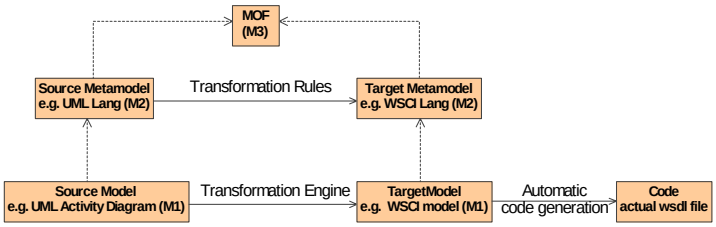

Figure 1: Transformation in MDA

Case Study Example

In order to illustrate, our Model Transformation approach, we shall present an example, which uses WSCI to model a business process. The example is a simplified version of an *"airline ticket reservation and booking"* business process, presented in WSCI specification. For the complete specification and example please refer to [17]. Here, the focus is on modelling the behaviour for only one of the participants involved; the *Traveller*. A UML Activity diagram that specifies the behaviour of the Traveller is presented. The model transformation converts such activity diagrams into an equivalent WSCI MOF based model [12], which is very close to the implementation code.

There are three Web services involved in the example; a *Traveller*, a *Travel Agent* and an *Airline System*. The business process *"Planning and Booking a Trip"* represents the interaction between the above three Web services as described in the follows scenario:

Initially, the *Traveller* plans a trip by deciding the destination, the departure and return date and time, and the preferred plan route. Then, she submits her request to the *Travel Agent.* The *Travel Agent* will evaluate the best itinerary for a destination and will build a proposed itinerary for the traveller. The proposed itinerary may not be satisfactory to the traveller, so she can submit a modified version and wait for a reply in the form of a new proposal by the *Travel Agent*. The *Traveller* can also choose to cancel the trip at any stage. In this case, the whole process terminates. However, if the plan is accepted, she has to reserve the tickets and provide her credit card details. If the reservation is confirmed, the *Traveller* can either book the tickets or cancel the reservation. There is a timeout for the booking period, i.e. tickets can be reserved for a limited time, at the end of which, the *Airline System* issues a notification and terminates the process.

At the end, the *Traveller* receives two set of items; an e-ticket from the e *Airline System* and a statement containing the charge details and the description of the trip itinerary from the *Travel Agent.* The process of *"Planning and Booking a Trip"* is completed upon the reception of both items.

UML Activity diagram

The process of *"Planning and Booking a Trip"* explained in the previous section can be modelled with a UML Activity diagram, see Figure 2, which represents both the coordination and interaction between the participants, either as a workflow or as choreography of message exchanges. The Activity diagram consists of three partitions, corresponding to three involving Web services. However, to reduce the size of the diagram, the Activities related to the *Traveller* and the flow of messages between the *Traveller* and the other two participants are only included.

The Activity diagram of Figure 2 for *"Planning and Booking a Trip"* can be explained as follows: The *Traveller* starts the process and triggers the message exchanges. Initially, she makes an order request to the *Travel Agent* by using a request-response WSDL operation called *OrderTrip.* In that case, the *TripOrderRequest* message is used to encapsulate the trip order details, where the *TripOrderAcknowledgement* message is the *Travel Agent's* response with the proposed itinerary.

If the *Traveller* is not happy with the itinerary, she may submit modifications by using the *ChangeItinerary* operation and the *ChangeItineraryRequest* message with the proposed itinerary. If it is accepted, a confirmation is returned. The whole process can be repeated as many times as the traveller is not satisfied with the plan. Notice, that such action is likely, only if it is triggered externally by the *Traveller* and *not* upon the arrival of a message or upon the satisfaction of a specific condition.

While there is an itinerary proposal, the *Traveller* has the choice to either reject the trip completely or accept it. If the trip is cancelled, the *Travel Agent* is notified by using the *CancelItinerary* operation with a *CancelItineraryRequest.* Then the *Travel Agent* will reply with a confirmation message. As a consequence, a *fault* will be generated that will terminate the process. If the plan is accepted, which is the default case, the *Traveller* has to reserve the tickets and provide her credit card information with a

ReserveTickets operation and a *ReservationRequest* message. Next, the *TravelAgent* confirms the reservation of the seats to the *Traveller*.

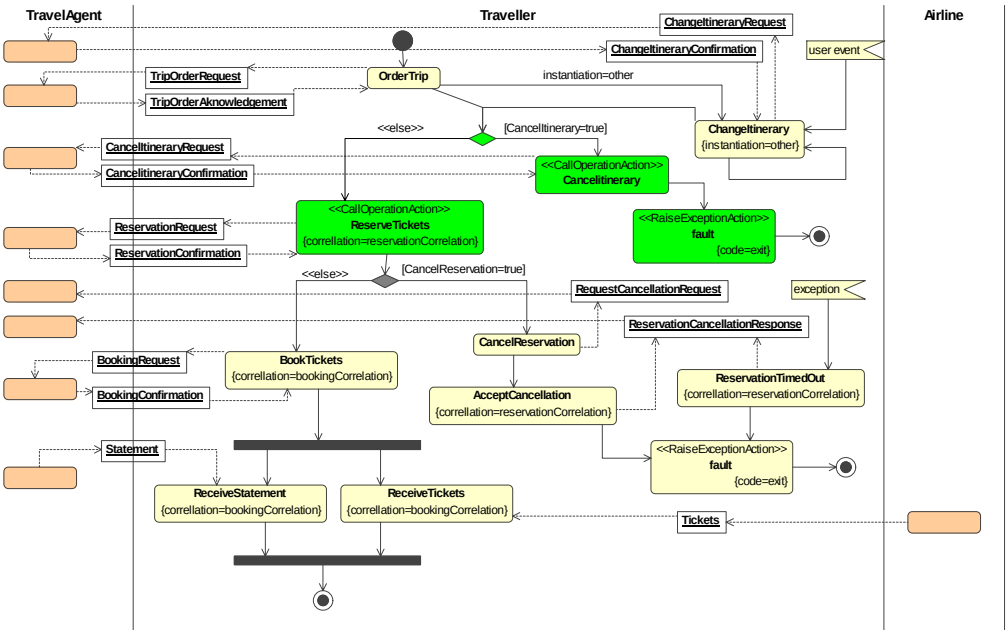

Figure 2: Activity diagram for Process "*Plan and Book Trip*"

Then, the *Traveller* can either book the tickets by the *BookTickets* operation and the *Booking Request* as the actual message passed or request a reservation cancellation with a *RequestCancellationRequest* message. In the first case, the *TravelAgent* will reply with a booking confirmation, where in the second case, the process will trigger the *AcceptCancellation* operation as a notification back to the *Airline* which as a result, it will generate an error and terminate. In the meantime, there is a limited period for reserving tickets, which if exceeded, a *ReservationTimeOut* exception is received. That in turn, it triggers again the *AcceptCancellation* operation and the process will be terminated as previously.

Afterwards, the *Traveller* waits to both receive from the *Airline* her e-ticket, encapsulated within a Ticket message through the *ReceiveTickets* operation, and from the *Travel Agent* a *statement* message, providing charge details and description of the planed trip, through a *ReceiveStatement* operation. The business process is then terminated.

Finally, one should notice the green (or grey in black and white) highlighted model elements, such as the *ReserveTickets* that are specific stereotyped and tagged, in order to provide additional information about their semantics. This information is necessary to direct an accurate transformation (as it will be demonstrated later) and can be regarded as a refined version, when compared with the rest of the modelling elements.

UML Metamodel

The following Figure 3, depicts a simplified UML metamodel, which includes UML activity and action, related model elements. We have compiled the metamodel from the UML Superstructure specification [15], containing the necessary modelling elements defined in the Actions and Activities sections.

Metamodels create a clear view of the available model elements, their relations and dependencies. In this case, the fundamental meta-elements are: *Activity* for expressing behaviour, *Action* for defining fundamental units of behaviour, *ActivityGroup* for defining sets of nodes within an *activity*, *ActivityPartition* for identifying actions with common characteristics, *ObjectFlow* for passing around objects or data, *ObjectNode* for the actual data, *ControlNode* which represents a class of various nodes used to coordinate flows in an activity, *ConditionalNode* for choosing among alternative options based on conditions, *CallOperationAction* for making an operation call and *StructuredActivityNode* for representing a structured portion within an activity. For further details regarding the model elements used, please refer to [15].

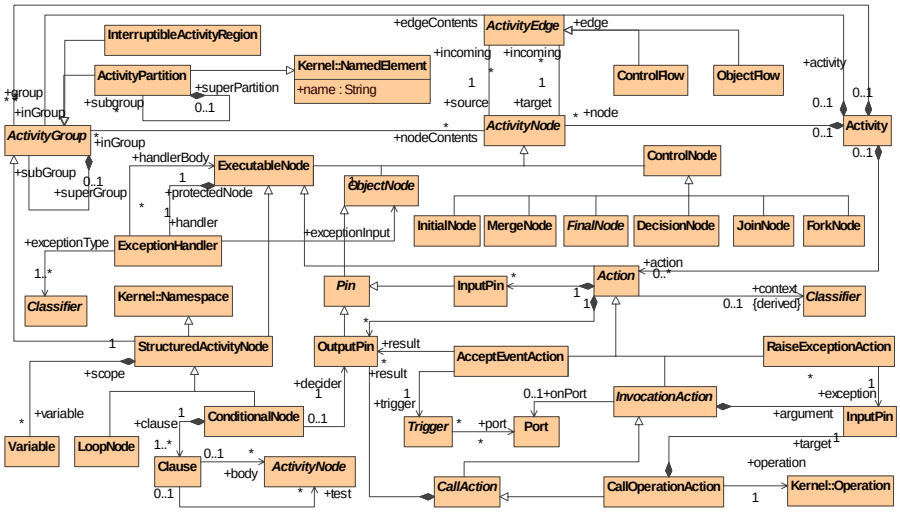

Figure 3: The UML Metamodel for Activities and Actions

WSCI Metamodel

This section presents the MOF metamodel for WSCI. The formal WSCI specification [17] makes use of the XML Schema mechanism [20]. As a result, the schema (standard mechanism) provides an XML based metamodel defining the framework's internal structure and semantics. However, since the MDA Model Transformation is in the context of MOF, we are interested in a WSCI metamodel that is MOF compliant. The following metamodel has been generated manually and is based on the W3C WSCI specification (version 1.0) [17].

The WSCI works in conjunction with the Web Service Description Language (WSDL) [18] and retains its respective semantics. The WSCI describes the dynamic Web service interfaces and reuses the operations and message types defined within the static interface parts (WSDL). For instance, *actions*, which are basic constructs of WSCI, describe how the service performs a fundamental activity as a WSDL operation.

The rest of this section, explains the metamodel of Figure 4. The *wsdl:definitions* element acts as a container for all top level WSCI definitions such as *interface*, *model* and *correlation.* The *model* describes how participants may interact through associated interfaces by connecting operations. The *correlation* describes how subsequent conversations are structured, by indicating a particular execution context (where the actions should be performed) and the *interface* provides the observable behaviour of a service by containing all the *process* definitions.

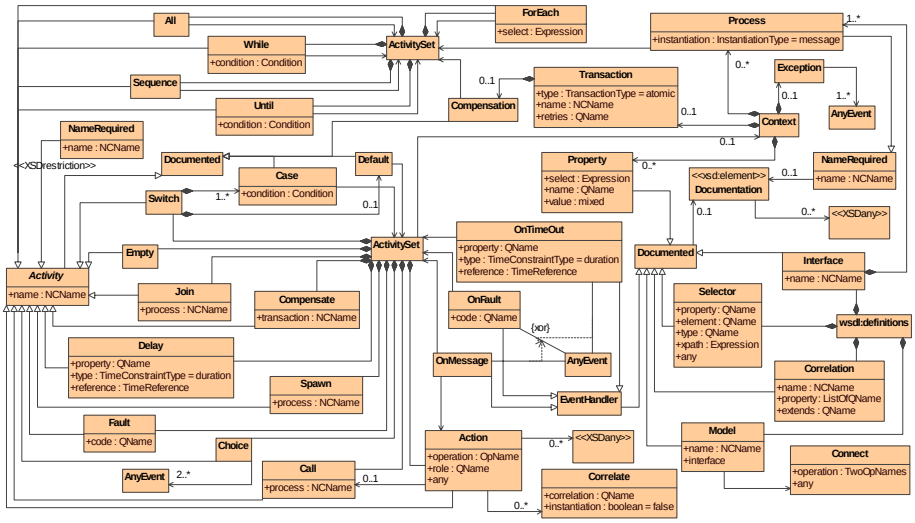

Figure 4: The MOF Metamodel for WSCI

The fundamental construct, which defines both simple and complex behaviour is the *activity* element. It forms the base type (abstract) for all other more elaborate *activity* elements. The activities can be distinguished into two conceptual categories, *atomic* and *complex*. The complex activities are composed from a number of activities defining various choreography effects. For instance, such activities may be performed in a sequential order (e.g. *sequence* element), after the validation of a condition (e.g *switch*) or in iteration (e.g. *while*). On the other side, atomic activities represent units of work (*actions*) that cannot be decomposed further. For example, the *call* activity triggers other *processes* and waits for them to complete while the *spawn* activity triggers other *processes* and continues without waiting them to complete.

A *process* is a special type of *activity* that establishes a *context* of execution, in which all activities are performed in a sequential order. The execution *context* defines the environment in which the activities should be performed and allows activities to share properties/variables. A *Property* is used to reference values from message types and are the equivalent of variables.

Finally, to simplify the metamodel, the authors have not incorporated the WSCI simple type elements, such as the "*TransactionType*" and "*Expression*".

Mapping UML Activity diagram To WSCI

The first step of the Model Transformation was to generate MOF compliant metamodels for both the source language and target language [6]. The next step is to identify corresponding model elements of the two metamodels. The following table depicts the correspondence between the model elements of metamodels of the UML Activity Diagram and WSCI. The shaded row is a model element from the source, UML Activity Diagram and the following un-shaded row is the corresponding model element from the destination, WSCI. The table also includes a description of each model elements based on the specifications provided in [15] and [17].

Activity Partition: groups together a set of activities that have common characteristics. They often correspond to organisation units, such as *Traveller* and *Travel Agent* in this case.
<interface>: contains the processes defining the behaviour of the Web service. For example, the behaviour of the *Traveller*.
ConditionalNode: represents an exclusive choice among a number of alternatives. It consists of one or more clauses. **DecisionNode:** is a control node that chooses between outgoing flows.
<switch> selects one activity set from a collection of two or more activity sets.
Clause's **Test** association: is a nested activity that specifies the result of the test.
<condition>: specifies a Boolean condition defined by XPath [19], that must evaluated.
Clause: represents a single branch of a condition construct including a test and a body section.
<case>: selects an activity set based on the truth value of a condition.
An <<**else**>> stereotyped **Clause**: is a clause that is a successor to all other clauses and where the test part always return true. It is similar to else in Java language.
<default>: selects an activity set in the event that no other condition has been met.
StructuredActivityNode: is an executable activity node. Subordinate nodes of such node *must* belong to only one *structured activity node*. The *activity* can start the execution when it has received its objects and control tokens.
<process>: is a special type of activity that establishes its own context of execution and where all activities are performed in sequential order.
CallOperationAction: An action that transmits an operation call. If the action is marked asynchronous the execution waits until the execution completes, otherwise it is completed when the invocation of the operation is established.
<action>: An atomic activity describing how an elementary WSDL operation can be used. It can be associated with either one-way, request response, notification and solicit response operations.
Operation association of **CallOperationAction**: The operation to be invoked.
operation: An attribute pointing at a WSDL operation. It can be associated with either one-way, request response, notification and solicit response operations.
ControlFlow: To model the sequencing of behaviours. **ObjectFlow:** To model the flow of values among activity nodes.
<sequence>: A complex activity that performs all its activities in sequential order.
StructuredActivityNode, InterruptibleActivityRegion: can be represented by a structured

activity node (accordingly stereotyped) and which it may contain an interruptible activity region.
<context>: It describes the environment in which a set of activities is executed.
RaiseExceptionAction: is an action that throws an exception.
<fault>: is an atomic activity that triggers a fault notification in the current context. This has an attribute *"code"*, which specifies the course of action in case that *fault* was triggered.
<<**correlate**>>**:** is a special stereotyped action.
<correlate>: is used to associate an action with a particular execution context in which the action should be performed.
ExceptionHandler: An element that specifies a body to execute in case a specified exception occurs. The exception type defines the instances the handler catches.
<exception>: An unexpected behaviour (event) is captured by defining an exception handler.
AcceptEventAction: An action that waits for the occurrence of an event meeting specified conditions. The type of event accepted, it is specified by a trigger.
<onMessage>: An event handler that responses to an incoming message. The initial action defines the event that triggers the handler.

Transformation Rules

In this section, we shall illustrate how the previous metamodel mapping can be applied, taking as an input the UML activity model of this case study and generating (output), either as an equivalent MOF [12] compliant WSCI model or as XML code. This is a type of an endomorphic transformation, as the source and target metamodels are MOF compliant. The rules introduced at this point, are for demonstrating purposes and are applied to a subset of modelling elements, which are highlighted in the source model as green, please refer to Figure 2. The rules can be materialised either by a specific transformation languages, OCL [13], OCL variations [5] or other languages [10].

In this case, the focus is on illustrating the principles behind the transformation rules, therefore we do not use a specific QVT [10][14] proposed language, but standard Object Constraint Language (OCL) statements, enhanced with a small number of conventions to support better the transformation process. These have been introduced on [1][9] and in this case are; the <~> to highlight the mapping rule and the try keyword following by the transformation rule to be performed. In addition, the statements are annotated to aid understanding. The transformation process involves a series of interlinked transformations, such as the UML *ConditionalNode* to a WSCI *Switch*, the UML *CallOperationAction* to a WSCI *Action*, the WSCI action correlation by a UML tagged property and the UML *RaiseExceptionAction* to a WSCI *Fault*. All these transformations have been named accordingly and are as follows:

Transformation UMLConditionalNode2WSCISwitch

```
--input parameters received
param input: UML::ConditionalNode
param output: WSCI::Switch
def attr cases: OCL::Set(UML::Clause) = input.clause
-- get all the CallOperation Actions within the clause's body
def oper getBodyActions(body: OCL::Set(UML::ActivityNode)):
OCL::Set(UML::CallOperationAction)= body->select(e: UML::ActivityNode |
e.oclIsTypeOf(UML::CallOperationAction))
-- get all the RaiseException Actions within the clause's body
def oper getBodyFaults(body: OCL::Set(UML::ActivityNode)):
OCL::Set(UML::RaiseExceptionAction)= body->select(e: UML::ActivityNode |
e.oclIsTypeOf(UML::RaiseExceptionAction))
--mapping
cases->iterate(c: UML::Clause |
if  c.oclIsTypeOf(<<else>>Action) then -– if it is a default case
    getBodyFaults(c.body)-> forAll(e: UML::RaiseExceptionAction |
    try UMLRaiseException2WSCIFault on e <~> output.fault )
else -– if it is a conditional case
  if c.test->forAll(e: InputPin | e.value==true) –-if condition is true
    output.case.condition <~> c
    getBodyActions(c.body)-> forAll(e: UML::CallOperationAction |
    try UMLCallOperation2WSCIAction on e <~> output.action )
  endif
endif )
```

Transformation UMLCallOperation2WSCIAction

```
--input parameters received
param input: UML::CallOperationAction
param output: WSCI::Action
def attr port: String = input.oclAsType(InvocationAction).onPort.name
--mapping
output.name <~> input.name
output.role <~> "tns:".concat(input.oclAsType(ActivityNode).inGroup->exists
                (t:ActivityPartion).oclAsType(ActivityPartition).name)
output.operation <~> "tns:".concat(port).concat("/").concat (input.operation.name)
post: -- any post conditions
--if there is a correlate attribute within the action
if input->attributes->exists(correlate)
  try UMLCorrelateAction2WSCICorrelate on input <~> output.correlate
endif
```

Transformation UMLCorrelateAction2WSCICorrelate

```
--input parameters received
param input: UML::CallOperationAction
param output: WSCI::Correlate
pre: --any pre conditions
input->attributes->exists(correlate)
--mapping
output.correlation <~> input.correlation –- is a tag value
if input->attributes->exists(instantiation)
  output.instantiation <~> input.instantiation – is a tag value
endif
```

Transformation UMLRaiseException2WSCIFault

```
--input parameters received
param input: UML::RaiseExceptionAction
param output: WSCI::Fault
--mapping
output.code <~> "tns:".concat(input.exception.name)
```

Generated Models & Code

If we apply the metamodel mapping on a full set of transformation rules, like the ones introduced on the previous section, a MOF based WSCI model can be derived from the initial case example, see Figure 5 bellow. The model is an instance of the WSCI metamodel presented on Figure 4 and it is stereotyped accordingly in order to illustrate the semantics of the classes.

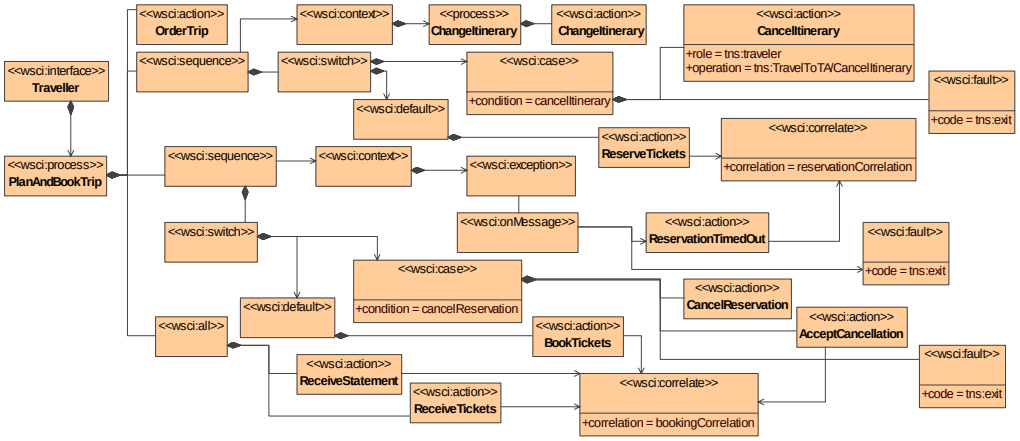

Figure 5: The MOF Based WSCI model produced

Equivalently, the following Figure 6 depicts a snippet of the code generated after applying the transformation rules. The code artefacts can be retrieved either from the WSCI model itself, as it encapsulates all the necessary information by performing a PSM to Code transformation (semantically they are very close), or directly, during the transformation process.

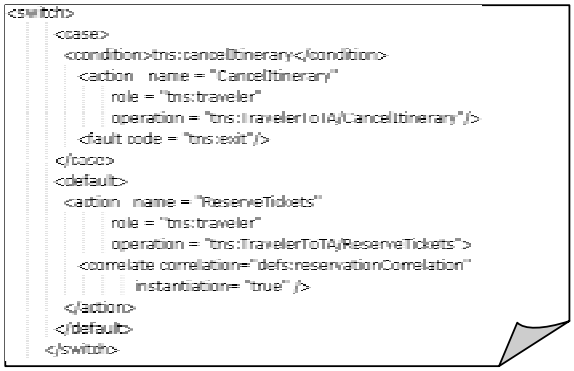

Figure 6: A snippet of the example code generated

Discussion / Conclusion

This work introduces the modelling, mapping and transformation of behavioural aspects of interacting Web services, within the context of MDA [8][3]. In particularly, we have demonstrated how a realistic scenario of Web service participants exchange messages in choreographed interactions via the WSCI standard. Initially, the behaviour of one of the participants was modelled with a UML activity diagram. Then the UML activity and WSCI metamodels are presented and a mapping among their equivalent meta-modelling elements is defined. Afterwards, a set of transformation rules in OCL was introduced to perform a transformation from the original UML model to a WSCI compliant MOF model and code. As a result, the whole process was successful and highlighted the applicability of MDA in transforming Web services standards, representing business processes, message interchanges and workflow.

Finally, the case study pointed out the initial idea; that certain types of systems and standards like the WSCI, BPEL etc are capable of capturing behaviour aspects, it is important to be modelled with rich behavioural models. This accommodates both their accurate design and mapping to their corresponding metamodels.

References

[1] A. Kleppe, J. Warmer, W. Bast, *MDA Explained. The Model Driven Architecture: Practice and Promise*, Addison-Wesley, ISBN: 321-19442-X, April 2003

[2] BEA, IBM, Microsoft, SAP AG and Siebel Systems, *Business Process Execution Language for Web Services (BPEL4WS)*, Version 1.1, May 2003

[3] D. Lopes, S. Hammoudi, *Web Services in the Context of MDA*, University of Nantes, France, 2003

[4] David Frankel, John Parodi, *White Paper: Using Model Driven Architecture to Develop Web Services*, IONA Technologies PLC, April, 2002

[5] J. Bezivin, E. Breton, G. Dupe, P. Valduriez, *The ATL Transformation-based Model Management Framework*, Research Report, Atlas Group, INRIA and IRIN, September 2003

[6] J. Bezivin, S. Hammoudi, D. Lopes, F. Jouault, *An Experiment in Mapping Web Services to Implementation Platforms*, Atlas Group, INRIA and LINA University of Nantes, Research Report, March 2004

[7] J. Siegel, *Developing in OMG's Model Driven Architecture*, Object Management Group, November 2002

[8] J. Siegel, *Using OMG's Model Driven Architecture (MDA) to integrate Web Services*, Object Management Group White Paper, Nov 2001

[9] J. Warmer, A. Kleppe, *The Object Constraint Language Second Edition*, Addison Wesley, ISBN 0321179366, August, 2003

[10] K. Czarnecki, S. Helsen, *Classification of Model Transformation Approaches*, OOPSLA'03 Workshop on Generative Techniques in the Context of Model-Driven Architecture, 2003

[11] M. Gogolla, A. Lindow, M. Richters, P. Ziemann,. *Metamodel Transformation of Data Models*, Position Paper, Proc. UML'2002 Workshop in Software Model Engineering (WiSME 2002)

[12] OMG, *Meta Object Facility (MOF) Specification*, Object Management Group, Version 1.4, April 2002

[13] OMG, *Object Constraint Language (OCL) Specification*, Object Management Group, Version 1.5, 2003

[14] OMG, *Request for Proposal: MOF 2.0 Query/View/Transformations RFP*, Object Management Group, 2002

[15] UML Superstructure OMG, *UML 2.0 Superstructure Spec.*, Object Management Group, Adopted Specification, 2003

[16] W3C, *Extensible Markup Language (XML) 1.0*, Third Edition, W3C Recommendation, February 2004

[17] W3C, *Web Service Choreography Interface (WSCI) 1.0*, W3C Note, August 2002

[18] W3C, *Web Services Description language (WSDL) Version 2.0*, W3C Working Draft, November 2003

[19] W3C, *XML Path Language (XPath) 2.0*, W3C Working Draft. July 2004

[20] XML Schema W3C, XML Schema Part 0: Primer, W3C Recommendation, May 2001