Experiment in Model Driven Validation of BPEL Specifications

D.H.Akehurst

University of Kent at Canterbury
dave@akehurst.net

Abstract. The Business Process Execution Language (BPEL) is an XML based language; the BPEL standard defines the structure, tags and attributes of an XML document that corresponds to a valid BPEL specification. In addition, the standard defines a number of natural language constraints of which some can be ambiguous and are complex. This paper uses the Unified Modelling Language (UML) and Object Constraint Language to provide a model of the XML based BPEL language. Based on this model the paper shows how OCL can be used to give a precise version of the natural language constraints defined in the BPEL standard. We then use this precise specification to generate a Validation tool automatically that can check that a BPEL document is well-formed.

1 Introduction

The problems of interoperability can be viewed as having two aspects: one being the matching of concepts and ontologies; the other being the problem of matching different technologies. In fact, most interoperability problems straddle this division, requiring both conceptual and technological compatibility to be addressed at the same time. If we can simplify one of these two aspects, we free up the engineering process to focus more fully on the other.

BPEL or BPEL4WS (Business Process Execution Language for Web Services) [4] is a language for specifying the Business Process logic that defines a choreography of interactions between a number of Web Services [9]. It is a technology that can be used to address aspects of interoperability between components that offer their services as Web Services. Although the language provides technical means to specify choreography patterns, it does not directly provide support for understanding the conceptual information associated with the related services. Making the use of this language as simple as possible via provision of good support tools will enable an engineer to focus on the conceptual issues rather than focusing on the difficulties of the language itself.

The BPEL4WS language is an XML based language and the BPEL standard defines the structure, tags and attributes of an XML document that corresponds to a valid BPEL specification. In addition to the precise specification of tag names and attributes, the standard defines a number of constraints on the way in which the XML elements should be put together. These constraints are given using natural language, which although being very descriptive is not always precise and some of the constraints are ambiguous. In addition some of the constraints are so complex that it is difficult to understand from the text what the constraint is actually saying; this leaves the possibility of creating what appears to be a valid XML BPEL document, which in actual fact violates one or more usage constraints.

This paper uses the Unified Modelling Language (UML) [8] and Object Constraint Language (OCL) [5] to provide a model of the XML based BPEL language. Based on this model the paper shows how OCL can be used to give a precise version of the natural language constraints defined in the BPEL standard.

The model and constraints are subsequently used to automatically generate a BPEL validation tool as a plug-in to IBM's eclipse IDE. The plug-in uses the Eclipse Modelling Framework (EMF) [3] code generation facilities along with the OCL code generation tools developed at the University of Kent [2].

Section 2 of this paper gives an explanation of the Model Driven approach to constructing the validator. Section 3 shows the UML model of the BPEL language (a BPEL metamodel) that we use. Section 4 specifies (some of) the OCL constraints that correspond to natural language from the BPEL4WS standard. Section 5 discusses some of the more difficult issues involved in mapping the model and constraints to an implementation. The paper concludes in section 6.

2 Explaining the Approach

Model Driven Development (MDD) or Model Driven Architecture (MDA) [7] is an approach to software development in which the focus is on Models as the primary artefacts. Model Transformations are considered the primary operation on models, used to map information from one model to another. One of the simplest instantiations of MDD is the idea of code generation from UML models, which can be viewed as a transformation from the UML model to a Code model.

Code generation is not new and has been around for far longer than the recent activity surrounding MDD. What we consider to be of interest with regards to the work in this paper is the combination of code generation from models and code generation from constraints; with the resulting code facilitating the validation of the constraints against instances of the model. In particular, the model code and the constraint code are generated separately, see Figure 1.

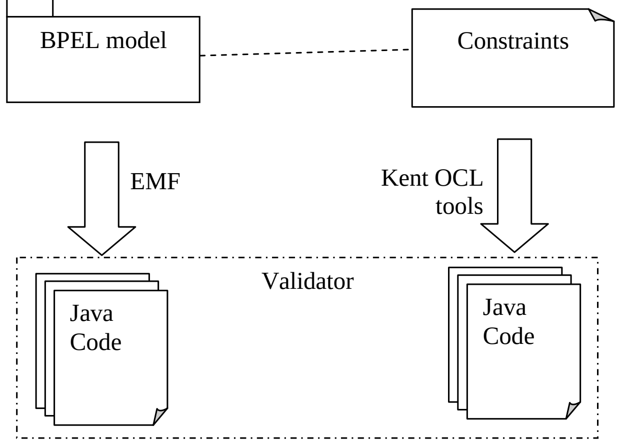


Figure 1 Code Generation from Model and Constraints

The work carried out in this paper is (as implied by the title) an experiment in using a modelling and constraint based approach to specifying the BPEL4WS language and automatically generating a tool that aids a user in writing valid BPEL specifications. This is different from the work described in [6] where UML is used as a language for writing BPEL specifications.

A set of UML Class Diagrams are created to model of the structural aspects of the BPEL language. We then use OCL as a mechanism for formally specifying constraints on instances of the model; these constraints on the model correspond to constraints on how elements of the BPEL language can be put together. A model constraint that fails would indicate an invalid combination of BPEL constructs.

OCL is a text based constraint language that has been officially part of the UML standard in recent versions. The language is based primarily on Set theory concepts and can be written using the ASCII character set. Although, defined initially as a "constraint" language the core expression part of OCL can also be used as an object-based Query Language.

3 BPEL Model

The BPEL language, as described above, is defined as an XML document. In order to write constraints using OCL we form a UML model that represents the structure of the BPEL XML document. The following figures illustrate part of the model, the rest can be found in the technical report [1]. The model of the BPEL language defined here is in accordance with the version 1.1 of the BPEL standard. There are however a few points to note regarding the modelling of the language:

- All yes/no options from the specification are mapped to Booleans; with 'true' representing 'yes'.
- The BusinessProcess class extends the class Scope. There is a large overlap between the two classes (partly due to changes from version 1.0 to 1.1), and the extension simplifies the model. The BusinessProcess class now contains PartnerLinks, these were not part of the BPEL 1.0 specification.
- The notion of 'imports' to a business process have been added (for the purpose of this work) in order to aid the reading of BPEL specifications and their reference to other documents.
- An additional Layer has been added to the activity hierarchy. The StructuredActivity and BasicActivity classer partitions the activities into those that contain sub-activities and those that don't. This helps with defining constraints to model the restrictions on links and the boundary crossing conditions.
- The class modelling the construct for a sequence of activities is renamed ActivitySequence (originally Sequence) as the original name clashes with the OCL type Sequence.
- PartnerLink is the version 1.1 replacement for the version 1.0 ServiceLinkType.

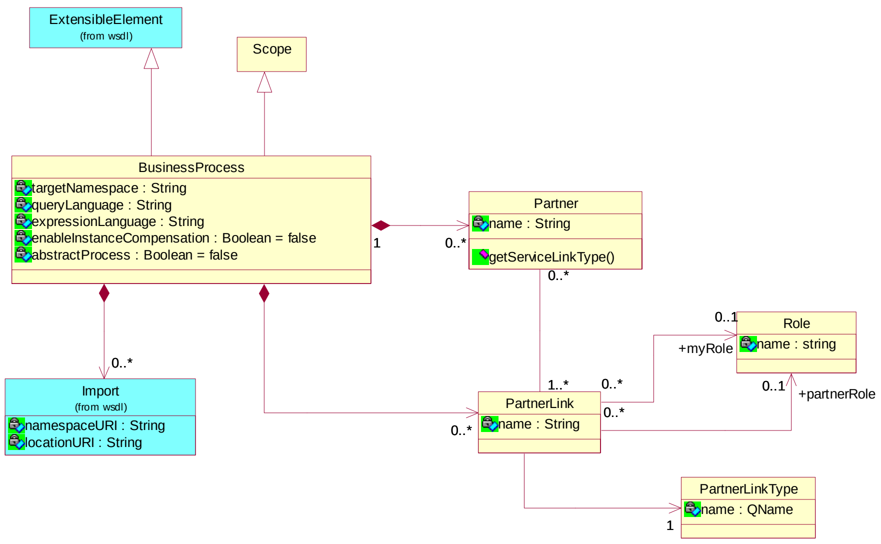


Figure 2 Business Process

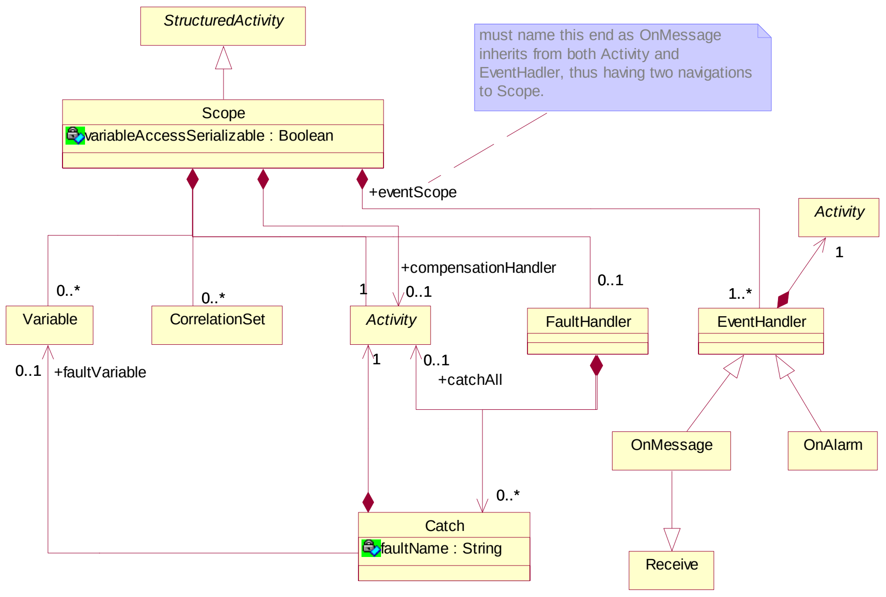


Figure 3 Scope

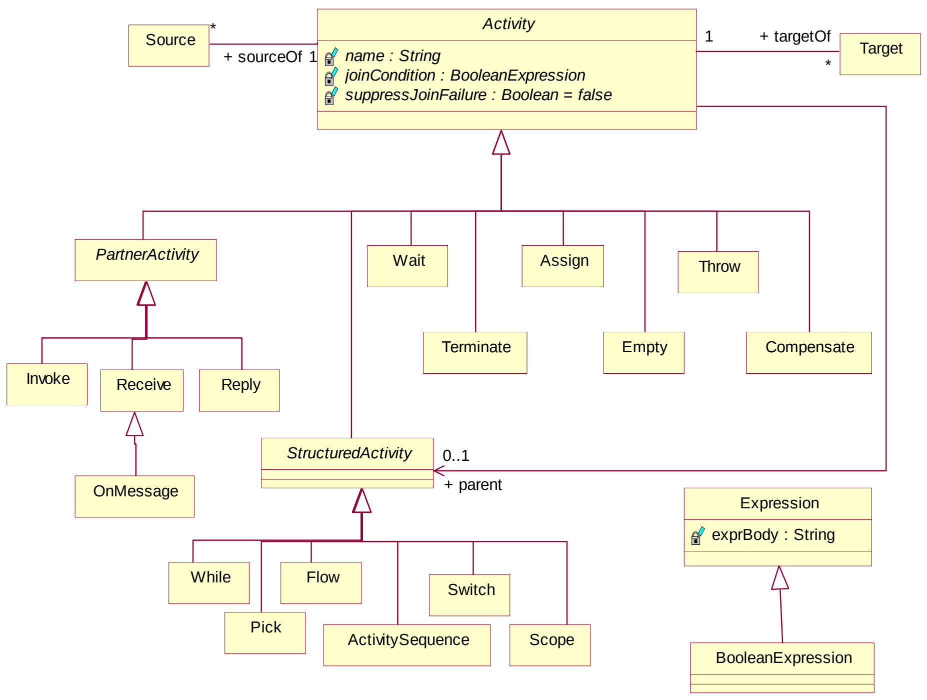


Figure 4 Activity Hierarchy and Standard Parts

4 The Constraints

The BPEL4WS standard contains approximately 20 natural language constraints that we have mapped into OCL constraints on the BPEL metamodel. The full details of this work can be found in the technical report [1], in this paper we have room to show only a few. We start by showing a simple constraints, then move on through successively more complex (and interesting) constraints until the final one that effectively require us to construct a BPEL model checker.

The following subsections address successively more complex constraints. Each of the subsections give an extract from the BPEL standard, defining a natural language constraint on the use of the language, which is then expressed using OCL to give a more precise specification as a constraint on the UML model of BPEL. The later constraints of greater complexity require the definition of additional model properties; OCL, through the use of the 'def' context, enables us to give these definitions, which can subsequently be used to specify the required OCL expressions.

4.1 Partner definitions must not overlap

The first constraint we look at is quite a simple constraint to define, however it does highlight an interesting issue regarding the relationship between the structural model of the language, OCL constraints on that model, and the implementation policy.

"7.3 Business Partners

… Partner definitions MUST NOT overlap, that is, a partner link MUST NOT appear in more than one partner definition."

This constraint is a restriction on the connections between a 'partner' construct and a 'partnerLink' construct. These two constructs are represented in BPEL as sub elements of the top level 'BusinessProcess' element. E.g. the following two XML segments show a valid and an invalid process specification:

```
<process
  name="Invalid" ...
 <partner
   name="SellerShipper"
   xmlns="http:...">
   <partnerLink
     name="Seller"/>
   <partnerLink
     name="Shipper"/>
 </partner>
 <partner
   name="Shipper">
   <partnerLink
     name="Shipper"
 ...
</process>
```

```
<process
  name="Valid" ...
 <partner
   name="SellerShipper"
   xmlns="http:...">
   <partnerLink
     name="Seller"/>
   <partnerLink
     name="Shipper"/>
 </partner>
 <partner
   name="Shipper">
   <partnerLink
     name="Shipper2"
 ...
</process>
```

This constraint requires that the union of partnerLink objects from all partners in a process is a Set; i.e. each partnerLink in that union is unique.

We could attempt to enforce this constraint by the structure of the BPEL model, however; i.e. if we model the relationship between Partner and PartnerLink as an [0..1]-to-[0..*] association, this states that any one partnerLink can only be associated to a single partner and thus the above constraint would not be violated. However, if we consider the generation of a BPEL validator, this is where the implementation policy is important; the structure of the XML language happily allows the constraint to be broken, so we must look at the process of mapping the XML document into an implementation of the BPEL model.

A typical implementation of a [0..1]-to-[0..*] association would simply update the [0..1] end if an object were added to the [0..*] end (this is certainly what happens within the EMF implementation). So a naïve mapping from XML to model implementation of the partner and partnerLink constructs would not provide any notification that the constraint was broken, it would simply update the links between instances.

For a BPEL validator, it is essential that notification is given that the constraint is violated. So either, the mapping from XML to model instance should check that the link between partner and partnerLink has not already been set; or we can model the association as a [0..*]-to-[0..*] association and add an explicit OCL constraint to check that the required uniqueness properties are met.

As can be seen by the UML diagrams above, we have adopted the second approach and the necessary OCL constraint is given below.

```
context bpel::Process
  inv partnerDefinitionsMustNotOverlap :
    let
    x=self.partner.partnerLink
  in
    x->asSet()->asBag() = x->asBag()
```

As can be seen by this example, there is a balance to be made between constraining the BPEL language by the structural model and by constraining it using OCL constraints.

4.2 Instantiation of a Process

This next constraint deals with the instantiation of a business process. This adds a constraint to the first actions that are allowed to occur in the process as a whole, and thus require us to extract a set of possible ‘initial activities’ from the complete activity graph define in the specification.

> “11.4. Providing Web Service Operations
>
> … The only way to instantiate a business process in BPEL4WS is to annotate a receive activity with the createInstance attribute set to "yes" (see 12.4. Pick for a variant). The default value of this attribute is "no". A receive activity annotated in this way MUST be an initial activity in the process, that is, the only other basic activities may potentially be performed prior to or simultaneously with such a receive activity MUST be similarly annotated receive activities.”

To check this constraint we first provide a property definition to extract the set of ‘initial activities’ for a BusinessProcess. The constraint is defined on ‘Scope’ a supertype of BusinessProcess to facilitate its reuse:

```
context bpel::Scope
def: initialActivities : Set(bpel::Activity) =
  self.activity.initialActivities
```

This property definition simply states that the initial activities of a BusinessProcess (or Scope) construct are the initial activities of the Activity construct that it contains. Thus we must also define the property ‘initialActivities’ as a property of ‘Activity’ and override the property in all subclasses of Activity in order to specify the correct definition for each type of activity. For reasons of space these overriding definitions are not given here, and the reader is directed to the technical report for full details.

We also provide a property ‘process’ on class ‘Activity’ that returns the business process containing the activity.

```
context bpel::Activity
  def: process : bpel::BusinessProcess =
    if self.parent.oclIsUndefined()
    then self.oclAsType(bpel::BusinessProcess)
    else self.parent.process endif
```

Given the definition of these properties we can provide two invariants that check the natural language constraint given above:

```
context bpel::Receive
 inv allReceivesMarkedAsCreateInstanceMustBeInitial :
  self.createInstance
   implies
  self.process.initialActivities->includes( self )
```

```
context bpel::BusinessProcess
 inv allInstantiationActivitiesAreMarkedCreateInstanceAsYes :
  self.instantiationActivities->forAll( act |
    act.oclIsTypeOf(bpel::Receive)
      and
    act.oclAsType(bpel::Receive).createInstance
  )
```

4.3 Link Control Cycles and boundary crossing

Some of the most complex constraints to check manually are those that define the use of Links within the Flow construct; i.e. those that check for boundary crossing conditions and control cycles. By using OCL to defining some additional (temporary) properties on the model we can construct OCL invariants that check the constraints.

Consider the following natural language constraints:

> "… Every link declared within a flow activity MUST have exactly one activity within the flow as its source and exactly one activity within the flow as its target. The source and target of a link MAY be nested arbitrarily deeply within the (structured) activities that are directly nested within the flow, except for the boundary-crossing restrictions.
>
> … In general, a link is said to *cross the boundary* of a syntactic construct if the source activity for the link is nested within the construct but the target activity is not, or *vice versa*, if the target activity for the link is nested within the construct but the source activity is not.
>
> … A link MUST NOT cross the boundary of a while activity, a serializable scope, an event handler or a compensation handler (see **13. Scopes** for the specification of event, fault and compensation handlers)."

We can illustrate a specification with Links that do and don't adhere to these constraints in Figure 5. The specifications are shown using a graphical notation rather than the XML syntax of BPEL as it is easier to see the links between activities in this manner. The arrows in the diagram represent links between activities. The top link is invalid as it crosses the boundary of a while activity; the lower link is valid.

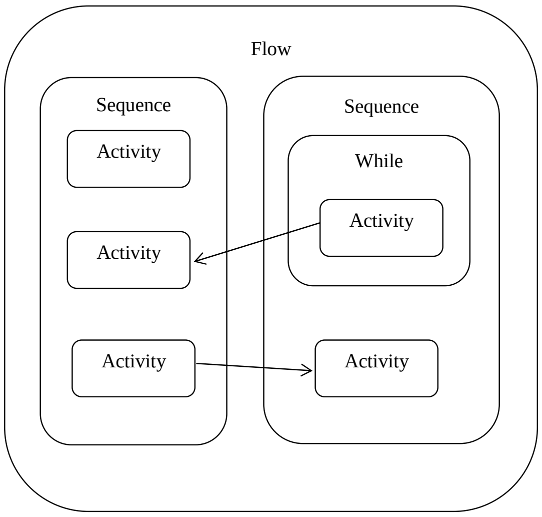


Figure 5 Links in a Flow Activity

To express this constraint in OCL, we require a property `subActivities` to be defined for each subtype of Activity. The property returns a set containing all activities directly nested within that Activity. Also required is a property `allSubActivities` which returns all nested and sub-nested activities. For basic Activities this set will typically be empty. These properties are defined for each Activity subtype in the technical report, overriding the property defined on their common supertype Activity as shown here:

```
context bpel::Activity
  def: subActivities : Set(bpel::Activity) = Set{}

  def: allSubActivities : Set(bpel::Activity) =
    self.subActivities
    ->union( self.subActivities.allSubActivities->asSet() )
```

The constraint requiring the source and target activity for each link of a flow to be contained within the flow can be expressed as follows:

```
context bpel_11::Flow
  inv sourceAndTargetActivitiesAreContainedWithinTheFlow :
    self.link->forAll( lnk |
      self.allSubActivities->includes( lnk.source.activity )
        and
      self.allSubActivities->includes( lnk.target.activity )
    )
```

We can subsequently test for boundary crossing violations on While, Scope and EventHandler constructs as follows:

```
context bpel::While
  inv boundryCrossing :
    self.allSubActivities->includesAll(
             self.allSubActivities.sourceOf.activity )
     and
    allSubActivities->includesAll(
             self.allSubActivities.targetOf.activity )

context bpel::Scope
  inv boundryCrossing :
    not self.variableAccessSerializable.oclIsUndefined()
      implies
   (self.variableAccessSerializable
      implies
    self.allSubActivities->includesAll(
                self.allSubActivities.sourceOf.activity )
      and
    self.allSubActivities->includesAll(
                self.allSubActivities.targetOf.activity )
    )

context bpel::EventHandler
  inv boundryCrossing :
   self.activity.allSubActivities->includesAll(
         self.activity.allSubActivities().sourceOf.activity )
     and
   self.activity.allSubActivities->includesAll(
         self.activity.allSubActivities().targetOf.activity )
```

Lastly, we check that links do not create control cycles in accordance with this natural language constraint:

"… Finally, a link MUST NOT create a control cycle, that is, the source activity must not have the target activity as a logically preceding activity, where an activity A logically precedes an activity B if the initiation of B semantically requires the completion of A. Therefore, directed graphs created by links are always acyclic."

This constraint requires us to construct expressions that enable navigation around a causality graph of the BPEL specification. To enable this we define properties that return the set of possible 'next' or 'previous' activities for any activity. Given the possibility to mix structured flow constructs (Switch, While, etc) with free form (Flow) constructs, it is not trivial to construct expressions that define 'next' and 'previous' properties.

If we define the property 'next', along with a transitive closure version 'allNext' that returns the set of all following activities, we can define a constraint that ensure links do not create control cycles as follows:

```
context bpel::Link
  inv noControlCycles :
    self.target.allNext->flatten()->excludes(self.source)
```

The complex part of this constraint is hidden in the expressions that form the 'next' properties, part of the definition for these properties is given below, the overriding versions for the other Activity subtypes can be found in the technical report:

```
context bpel::StructuredActivity
  def: next(prev:bpel::Activity) : Set(Activity) =
    Set{}

context bpel::Activity
  def: next : Set(bpel::Activity) =
   let
    x =  self.parent.next(self)
   in
    if x->isEmpty()
    then self.parent.next
    else x endif

  def: allNext : Sequence(Set(bpel::Activity)) =
    Sequence{Set{self}}->transitiveClosure(x|x.next->asSet())

context bpel::While
  def: next(prev:bpel::Activity) : Set(Activity) =
    self.activity.initialActivities
    ->union( self.parent.next )
```

Note here that we require a transitive closure operation in order to define the 'allNext' property. It would potentially be possible to manage without it; however the definition of 'allNext' would require the use of additional recursive operations and be more complex to define.

4.4 Correlations between Receive and Reply Activities

By far the most complex constraints described using natural Language are those that discuss the relationship between receive and reply activities that use the same correlation sets, operations and ports. A correlation set is a mechanism for identifying a particular instance of a remote statefull service. These constraints address the situation of receiving a message from just such a remote service and the subsequent reply to that service. They attempt to reduce the possibility of causing deadlocks and inconsistencies due to miss matching of receive-reply pairs.

The following is one of the constraints addressing this issue:

> "… The correlation between a request and the corresponding reply is based on the constraint that more than one outstanding synchronous request from a specific partner link for a particular portType, operation and correlation set(s) MUST NOT be outstanding simultaneously. The semantics of a process in which this constraint is violated is undefined. ..."

To construct an OCL constraint to check that this situation does not occur, it is necessary to form an expression that first collects the sequence of activities that may possibly follow a receive activity; we can potentially use the previously define 'next' and 'allNext' properties to do this. This sequence of following activities must then be searched for the first matching reply activity, and finally an expression needs to formed to check that no other matching receive activity occurs in-between the first receive activity and first subsequent matching reply activity. The following OCL invariant forms the required constraint:

```
context bpel_11::Receive
  inv noSimultaneousOutstandingSynchronousRequests  :
    let
      request = self,
      actionSeq = self.allNext->flatten(),
      replySeq = actionSeq->select( a |
                     a.oclIsTypeOf(bpel_11::Reply)
                  ).oclAsType(Sequence(bpel_11::Reply)),
      reply = replySeq->select( r |
                  r.portType= request.portType and
                  r.operation = request.operation and
                  r.correlation = request.correlation
                )->first(),
      replyIndex = actionSeq->indexOf(reply)
    in
      not actionSeq->subsequence(0,replyIndex)
          ->select(a|a.oclIsTypeOf(bpel_11::Receive))
          .oclAsType(Sequence(bpel_11::Receive))
          ->exists( rec |
           rec.portType= request.portType and
           rec.operation = request.operation and
           rec.correlation = request.correlation
          )
```

Unfortunately this constraint does not fully perform an accurate check. The semantics of the Link construct cause execution to block at the target of a link until the action at the source of a link has completed. The defined property 'next' on the Flow activity does not take into consideration the full link-semantics of the language; thus causing the constraint to falsely fail under conditions that are in actual fact not a problem. To provide a fully accurate check the definition of the 'next' property would need to take into consideration the blocking semantics of links between activities that are nested within a Flow activity. This would amount to providing a true flow analysis of the activity structure and although it may be possible to do this using OCL, the complexity of the expressions comes close to constructing a model checking algorithm, and we consider this to be outside the scope of appropriate use of OCL.

4.5 getLinkStatus Function

There are a couple of constraints in the BPEL standard that apply to the build in function `"bpws:getLinkStatus('linkName')"`; e.g.

> "9.1 Expressions
>
> … [bpws:getLinkStatus ('linkName')] function MUST NOT be used anywhere except in a join condition. The linkName argument MUST refer to the name of an incoming link for the activity associated with the join condition. These restrictions MUST be statically enforced."

To check this constraint using OCL it would require us to parse the contents of the 'expression' attribute on a number of BPEL constructs. We feel that to construct such a parsing algorithm using OCL would most definitely be inappropriate in the context of adding constraints to a structural model.

5 Generating a BPEL Validator

It is all very well to draw a few diagrams containing boxes, lines and snippets of text and then write constraints in a funny kind of ASCII compatible set theory – but what use is it?

Firstly, it does allow us to define unambiguously and precisely the usage constraints of the language.

Secondly, and even more usefully, we can automatically generate a validation tool that will check that the constraints have been adhered to, for any particular language expression (in this case, any particular BPEL document).

Using one of the UML (or MOF or ECORE) model to code automatic generation tools, such as the Eclipse Modelling Framework (EMF), we can quickly and easily generate a tool that supports a model based representation of a BPEL specification.

At the University of Kent we have developed an OCL library that operates in conjunction with EMF repositories to facilitate both the evaluation of OCL constraints and the generation of code that corresponds to an OCL constraint.

Thus, using these two tools (EMF and Kent OCL) the core code for a BPEL validator can be generated. The code can then be wrapped up to give the required user interface; for example we have wrapped up the validator as an eclipse plug-in, see **Error! Reference source not found.**.

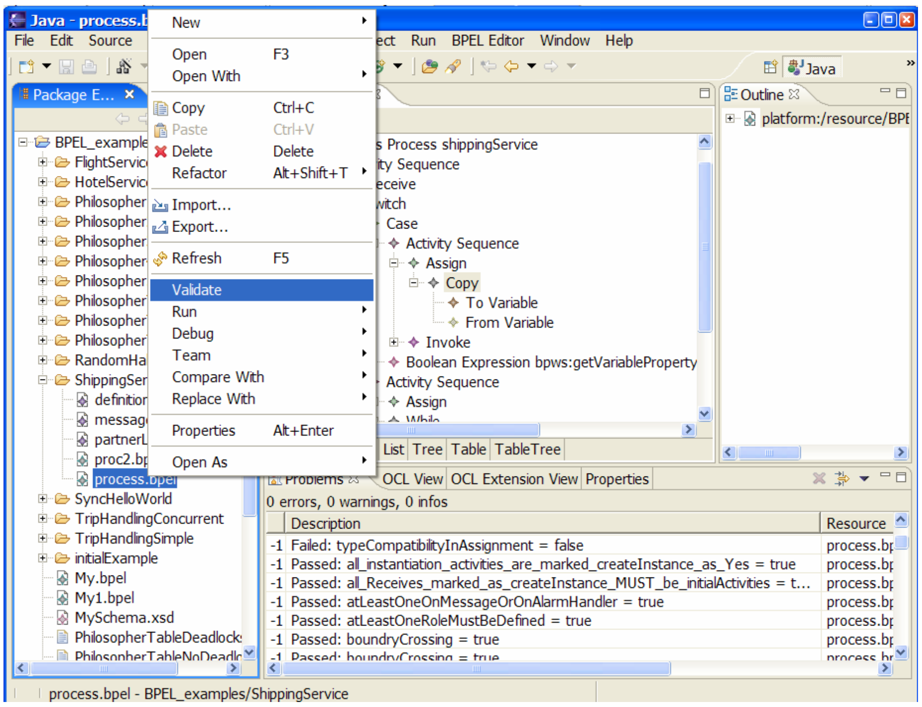
There are two mechanisms provided by the Kent OCL library for evaluating OCL expressions. There is an interpreter, that directly evaluates an expression and there is a code generation interface that converts the OCL expression into equivalent Java code. For the BPEL validator we use the Java code generation interface, as this allows the validator to be used independently from the OCL library. Constructing the BPEL validator has helped to develop the code generation facility of the Kent OCL library.

There are several issues worth drawing out regarding our experiences of mapping OCL expressions to Java code, in addition to the highlighting of inconsistencies in the specification of the OCL standard; below we discuss a couple of the more interesting OCL-to-Java issues.

The first issue we consider worth mentioning is the mapping of Boolean expressions between OCL and Java. The Java language uses true two-value Boolean logic, whereas the Boolean logic of OCL is actually based on three-value logic; i.e. *true*, *false*, and *undefined*. Providing a mapping from the three-value OCL to the two-value Java is thus not a simple process; in addition, Java expressions that would typically cause an exception (e.g. a NullPointerException), in OCL simply evaluate to the *undefined* value. Consequently, the Java code for an OCL Boolean expression must first catch exceptions for each sub-part of the expression, and then use a structure of conditional statements to convert the two-value semantics of Java into the correct interpretation of the OCL three-value semantics.

The second issue we illustrate here is that of providing support for the 'def' context of OCL. This part of the OCL language is used to define additional properties and operations on Classes and Types defined by the structural part of a UML model. We have seen the use of this in the constraints specified in the previous section.

The difficulty here is that the Java code for the model types is generated by the EMF toolkit; however we wish to add additional properties and operations as a separate process. One option is to alter the code generated by EMF, however this is not ideal as it would require parsing and understanding the generated code;



```
0 errors, 0 warnings, 0 infos
   Description                                                                   Resource
-1 Failed: typeCompatibilityInAssignment = false                                 process.bp
-1 Passed: all_instantiation_activities_are_marked_createInstance_as_Yes = true  process.bp
-1 Passed: all_Receives_marked_as_createInstance_MUST_be_initialActivities = t... process.bp
-1 Passed: atLeastOneOnMessageOrOnAlarmHandler = true                            process.bp
-1 Passed: atLeastOneRoleMustBeDefined = true                                    process.bp
-1 Passed: boundryCrossing = true                                                process.bp
-1 Passed: boundryCrossing = true                                                process.bp
```

Figure 6 BPEL Validator as an eclipse plug-in

additionally in a more general scenario we might not have access to the source code of the model types. Ideally we require a mechanism that can be used to 'semantically' add operations and properties (from the perspective of the OCL code) without touching the implementation of the model types.

Our solution to this has been to generate static methods that represent the 'defined' properties and operations. When generating code for an OCL expression that calls one of these 'defined' features rather than calling a method on the source object, the source object is passed as a parameter to the static method. To facilitate the overriding semantics, the static method itself is not called directly, but called via a special method that looks for a parameter match on the closest runtime type of the source object.

6 Conclusion

In this paper we have shown the use of OCL and UML to provide a precise version of natural language constraints on the structuring of BPEL XML documents. From these precise specifications of the constraints we can automatically build a validator to check that the constraints have been met for any example BPEL document. This is particularly useful in the case of BPEL as some of the natural language constraints are ambiguous or complex to understand.

We have discussed in this paper a number of consecutively more complex forms of constraint: those that can be formed directly from OCL expressions; those that require the addition of extra properties; and those that require complex algorithms.

In particular, the constraint discussed in section 4.1 highlights an interesting requirement to balance constraints impose indirectly by the structure of a model, explicit constraints added using OCL and the implementation strategy involved. The later constraints of sections 4.4 and 4.5 illustrate a limit to what we consider appropriate use of OCL and the constraints in section 4.3 illustrate the requirement for a transitive closure operation to be added to OCL.

Finally the paper has discussed some of the issues regarding the automatic generation of code, in this case a BPEL validator, using OCL constraints as the source.

References

[1] Akehurst D. H., "Validating BPEL Specifications Using OCL," University of Kent at Canterbury, technical report: 15-04, August 2004.

[2] Akehurst D. H. and Patrascoiu O., "OCL 2.0 – Implementing the Standard for Multiple Metamodels," in proceedings UML 2003 Workshop, OCL 2.0 - Industry standard or scientific playground?, San Francisco, USA, October 2003.

[3] Eclipse.org, "Eclipse Modelling Framework (EMF)," www.eclipse.org/emf

[4] IBM, "Business Process Execution Language for Web Services," 2003, www.ibm.com/developerworks/library/ws-bpel/

[5] Kleppe A. and Warmer J., "The Object Constraint Language and its application in the UML metamodel," in proceedings <<UML>>'98 Beyond the Notation, Mullhouse, France, June 1998.

[6] Mantell K., "From UML to BPEL," 2003, http://www-106.ibm.com/developerworks/webservices/library/ws-uml2bpel/

[7] OMG, "Model Driven Architecture (MDA)," Object Management Group, ormsc/2001-07-01, July 2001.

[8] OMG, "The Unified Modeling Language Version 1.5," Object Management Group, formal/03-03-01, March 2003.

[9] W3C, "Web Services Definition Language (WSDL) 1.1," 2001, http://www.w3.org/TR/wsdl