TUHH
Technische Universität Hamburg-Harburg

STS

BPEL Validation with Object Constraint Language (OCL)

Student Project

Submitted by:

Madanagopal Doraiswamy Venkatesan
madanagopal.doraiswamy@tu-harburg.de
Matriculation Number: **27125**

supervised by

Prof Dr. Ralf Möller,
M.Sc. Miguel Garcia,
Software, Technology & Systems (STS),
Hamburg University of Technology,
Germany.

Declaration

I declare that:
this work has been prepared by myself,
all literal or content based quotations are clearly pointed out,
and no other sources or aids than the declared ones have been used.

Hamburg, 30-09-2005.

Madanagopal Doraiswamy Venkatesan

Acknowledgements

I would like to extend my sincere gratitude to Prof. Dr. Ralf Möller for giving me an oppurtunity to do my Student Project at Software, Technology & Systems (STS) Department of Hamburg University of Technology, Germany.

I would like to thank M.Sc. Miguel Garcia at Software, Technology & Systems (STS) Department for introducing me to this topic and giving his support, guidance and valuable suggestions throughout my project work.

Finally, I would like to thank my friends for their continuous support and encouragement, especially in encouraging me to use LaTeX.

Abstract

Business Process Execution Language (BPEL), being a XML based language; its standard defines the structure, tags and attributes of the 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. My project 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. With this precise specification a Validating tool could be developed to automatically check the BPEL document for its well-formedness. Development of such validating tools would help developers to focus on implementation of business process rather than focussing on writing a well-formed BPEL document.

Contents

1 Introduction **1**

2 BPEL4WS **2**
2.1 BPEL . . . 2
2.2 Structure of BPEL 1.1 [3] . . . 2
2.3 BPEL 1.1 Metamodel . . . 4
2.3.1 Business Process: . . . 4
2.3.2 Scope, Variable and CorrelationSet: . . . 4
2.3.3 Activity Hierarchy and Standard Parts: . . . 5
2.3.4 Assign Activity: . . . 5
2.3.5 Structured Activities: . . . 5
2.3.6 WSDL Extensions: . . . 5

3 Need for Constraints **13**
3.1 Introduction to OCL . . . 13
3.2 Types of Constraints . . . 14
3.3 Why OCL? . . . 14
3.4 Constraints for BPEL [1] . . . 14
3.4.1 Business Process . . . 15
3.4.2 Partner Definitions must not overlap [2] . . . 15
3.4.3 Variable Options . . . 16
3.4.4 Source and Target of Activities . . . 17
3.4.5 Pick . . . 17
3.4.6 Flows and Links . . . 17

4 Java Architecture for XML Binding (JAXB) **19**
4.1 What is JAXB? . . . 19
4.2 Comparison with JAXB 1.0 . . . 19
4.3 How Does JAXB fit in our case? . . . 20
4.3.1 BPEL Schema . . . 20
4.3.2 Binding Declarations . . . 20
4.3.3 Binding Compiler . . . 20
4.3.4 Binding Framework Implementation . . . 21
4.3.5 Schema-Derived Classes . . . 21
4.3.6 BPEL Validation Application . . . 21
4.3.7 BPEL Input Documents . . . 21
4.3.8 XML Output Documents . . . 21
4.4 JAXB 2.0 Binding Process for BPEL . . . 21
4.4.1 Generate Classes . . . 21

4.4.2 Compile Classes 22
4.4.3 Unmarshal 22

5 Validation **25**
5.1 Performing a BPEL Validation with OCL 25

6 Conclusion **29**

7 Appendix A **30**
7.1 Defined Properties and Operations - OCL Constraints 30
7.1.1 Define Property - subActivities : Set 30
7.1.2 Define Property - allSubActivities : Set 31
7.1.3 Definition of Property - initialActivities : Set 31
7.1.4 Definition of Property - allBasicActivities : Set 32
7.1.5 Definition of Property - next : Set 33
7.1.6 Definition of Property - orderedSubActivities : Set 33
7.1.7 Definition of Property - allOrderedSubActivities : Set 34
7.1.8 Definition of Property - allParents : Set 34
7.1.9 Definition of Property - process : BusinessProcess 35
7.1.10 Definition of Property - instantiationActivities : Set 35
7.1.11 Definition of Property - causalGroups : Set 35

8 Appendix B **36**
8.1 Generics 36
8.1.1 Introduction 36
8.1.2 WildCards in JAXB 2.0 37

9 Appendix C **38**
9.1 Octopus - OCL Tool for Precise Uml Specifications 38
9.1.1 Introduction 38
9.1.2 UML/OCL Transformation to Code 38

List of Figures

2.1 Business Process 6
2.2 Scope, Variable and CorrelationSet 7
2.3 Activity Hierarchy 8
2.4 Assign Activity 9
2.5 WSDL Extensions, Basic Activities 10
2.6 Partner Activities 11
2.7 Structured Activities 12

3.1 Invalid and Valid XML Segment 16

4.1 General JAXB Overview 20
4.2 Core JAXB Components 22
4.3 JAXB Content Object 22
4.4 Content Tree for AmazonFlow.bpel 24

5.1 Code Generation from Schema, Metamodel and Constraints 26
5.2 Hierarchy of Activity 27

Chapter 1

Introduction

BPEL [3] or BPEL4WS[1] is a language for specifying the Business Process logic that defines a choreography of interactions between a number of Web Services. It provides a language for the formal specification of business processes and business interaction protocols. 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 validation tools will enable an engineer to focus on the conceptual issues rather than focusing on the difficulties of the language itself.

This paper uses the UML[2] and OCL[3] 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.

Section 2 of this paper gives an explanation of BPELWS metamodel(UML Model) and its structural aspects. Section 3 specifies about a brief introduction to OCL Constraints ,its use in designing a validator and about the OCL constraints employed in validating the Constraint Code against the instances of BPEL. Section 4 discusses about JAXB 2.0 [9], the new version of Java Architecture for XML Binding and its implementation in this project. Section 5 discusses about the steps taken in creating the validation for BPEL with some difficult issues involved in mapping the JAXB generated objects for BPEL Schema with that of the OCTOPUS [6] generated Constraint Code for BPEL Metamodel. Section 6 specifies the future enhancements that could be done to this project. The paper concludes with section 7.

[1] Business Process Execution Language for Web Services
[2] Unified Modelling Language
[3] Object Constraint Language

Chapter 2

BPEL4WS

2.1 BPEL

BPEL4WS provides a language for the formal specification of business processes and business interaction protocols. By doing so, it extends the Web Services interaction model and enables it to support business transactions. Business processes can be described in two ways. Executable business processes, model the actual behavior of a participant in a business interaction. Abstract processes, in contrast, specify the mutually visible message exchange behavior of each of the parties involved in the protocol, without revealing their internal behavior. BPEL4WS [3] is meant to be used to model the behavior of both executable and abstract processes. In other words, BPEL may be used to define the external behavior of a service (with an abstract process) as well as the internal implementation (with an executable process).

BPEL fits into the core Web service architecture since it is built on top of XML, XML Schema, WSDL, and UDDI. All external resources and partners are represented as WSDL services.

2.2 Structure of BPEL 1.1 [3]

The Structure of BPEL 1.1 starts with the top-level attributes:

- **queryLanguage:** This attribute specifies the XML query language used for selection of nodes in assignment, property definition, and other uses. The default for this attribute is XPath 1.0, represented by the URI of the XPath 1.0 specification.
- **expressionLanguage:** This attribute specifies the expression language used in the process. The default for this attribute is XPath 1.0, represented by the URI of the XPath 1.0 specification.
- **suppressJoinFailure:** This attribute determines whether the joinFailure fault will be suppressed for all activities in the process. The effect of the attribute at the process level can be overridden by an activity using a different value for the attribute. The default for this attribute is “no”.
- **enableInstanceCompensation:** This attribute determines whether the process instance as a whole can be compensated by platform-specific means. The default for this attribute is “no”.

- **abstractProcess:** This attribute specifies whether the process being defined is abstract (rather than executable). The default for this attribute is "no".

The four major sections in a business process definition are:

- <**variables**> section defines the data variables used by the process, providing their definitions in terms of WSDL message types, XML Schema simple types, or XML Schema elements. Variables allow processes to maintain state data and process history based on messages exchanged.

- <**partnerLinks**> section defines the different parties that interact with the business process in the course of processing the order. Each partner link is characterized by a partner link type and a role name. This information identifies the functionality that must be provided by the business process and by the partner service for the relationship to succeed, that is, the portTypes that the process and the partner need to implement.

- <**faultHandlers**> section contains fault handlers defining the activities that must be performed in response to faults resulting from the invocation of the assessment and approval services.

- The rest of the process definition contains the description of the normal behavior for handling any business process request. The elements of this description has to be any of the types of below listed "activity".

  - <**receive**> construct allows the business process to do a blocking wait for a matching message to arrive.
  - <**reply**> construct allows the business process to send a message in reply to a message that was received through a <**receive**>. The combination of a <**receive**> and a <**reply**> forms a request-response operation on the WSDL portType for the process.
  - <**invoke**> construct allows the business process to invoke a one-way or request-response operation on a portType offered by a partner.
  - <**assign**> construct can be used to update the values of variables with new data. An <**assign**> construct can contain any number of elementary assignments.
  - <**throw**> construct generates a fault from inside the business process.
  - <**wait**> construct allows to wait for a given time period or until a certain time has passed. Exactly one of the expiration criteria must be specified.
  - <**empty**> construct allows to insert a "no-op" instruction into a business process. This is useful for synchronization of concurrent activities.
  - <**sequence**> construct allows to define a collection of activities to be performed sequentially in the order they are listed.
  - <**switch**> construct includes a set of activities, each associated with a condition. The activity associated with the first true condition is executed, while the others are skipped. It is also possible to specify an otherwise activity, executed if no condition is true.
  - <**while**> construct allows to indicate that an activity is to be repeated until a certain success criteria has been met.

- <**pick**> construct allows to block and wait for a suitable message to arrive or for a time-out alarm to go off. When one of these triggers occurs, the associated activity is performed and the pick completes.
- <**flow**> construct allows to specify one or more activities to be performed concurrently. Links can be used within concurrent activities to define arbitrary control structures.
- <**scope**> construct allows to define a nested activity with its own associated variables, fault handlers, and compensation handler.
- <**compensate**> construct is used to invoke compensation on an inner scope that has already completed normally. This construct can be invoked only from within a fault handler or another compensation handler.

2.3 BPEL 1.1 Metamodel

Modeling of BPEL is required to facilitate the writing of OCL constraints and generating validating code with Octopus[1]. There are however a few points to note regarding the modeling of the language.

2.3.1 Business Process:

- 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 contains PartnerLinks, and these were not part of the BPEL 1.0 specification.
- Code Generation in Octopus fails to generate for classes with the names of Java reserved words. So such classes have been renamed with their original names appended with "BPEL" string. For Example, Classes like Switch, Catch are renamed as SwtichBPEL and CatchBPEL respectively.
- 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.
- ServiceLinkType of BPEL 1.0 is being replaced with PartnerLink, in the current version.

2.3.2 Scope, Variable and CorrelationSet:

The Scope Construct as shown in Figure 2.2 contains a collection of Variables (a replacement for Containers), a collection of CorrelationSets and an optional EventHandler.

The Variable refers to one of the three optional parts, Message, SimpleType, Element. An OCL constraint is implemented to restrict the reference to one of these parts.

[1] Refer to Appendix C

2.3.3 Activity Hierarchy and Standard Parts:

An additional Layer has been added to the activity hierarchy as shown in Figure 2.3. The StructuredActivity and BasicActivity classes partitions the activities into those that contain sub-activities and those that do not. This helps with defining constraints to model the restrictions on links and the boundary crossing conditions.

2.3.4 Assign Activity:

The alternative options for From and To specs in the Copy construct are modelled as sub-types.

2.3.5 Structured Activities:

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 as shown in Figure 2.7.

The OnAlarm construct is modeled with a single expression and an enumeration to indicate whether it is an “until” or “for” expression; rather than two exclusive - or expression attributes.

2.3.6 WSDL Extensions:

PartnerLink is the version 1.1 replacement for the version 1.0 ServiceLinkType.. All WSDL Extension elements have been appened with BPEL with their names.

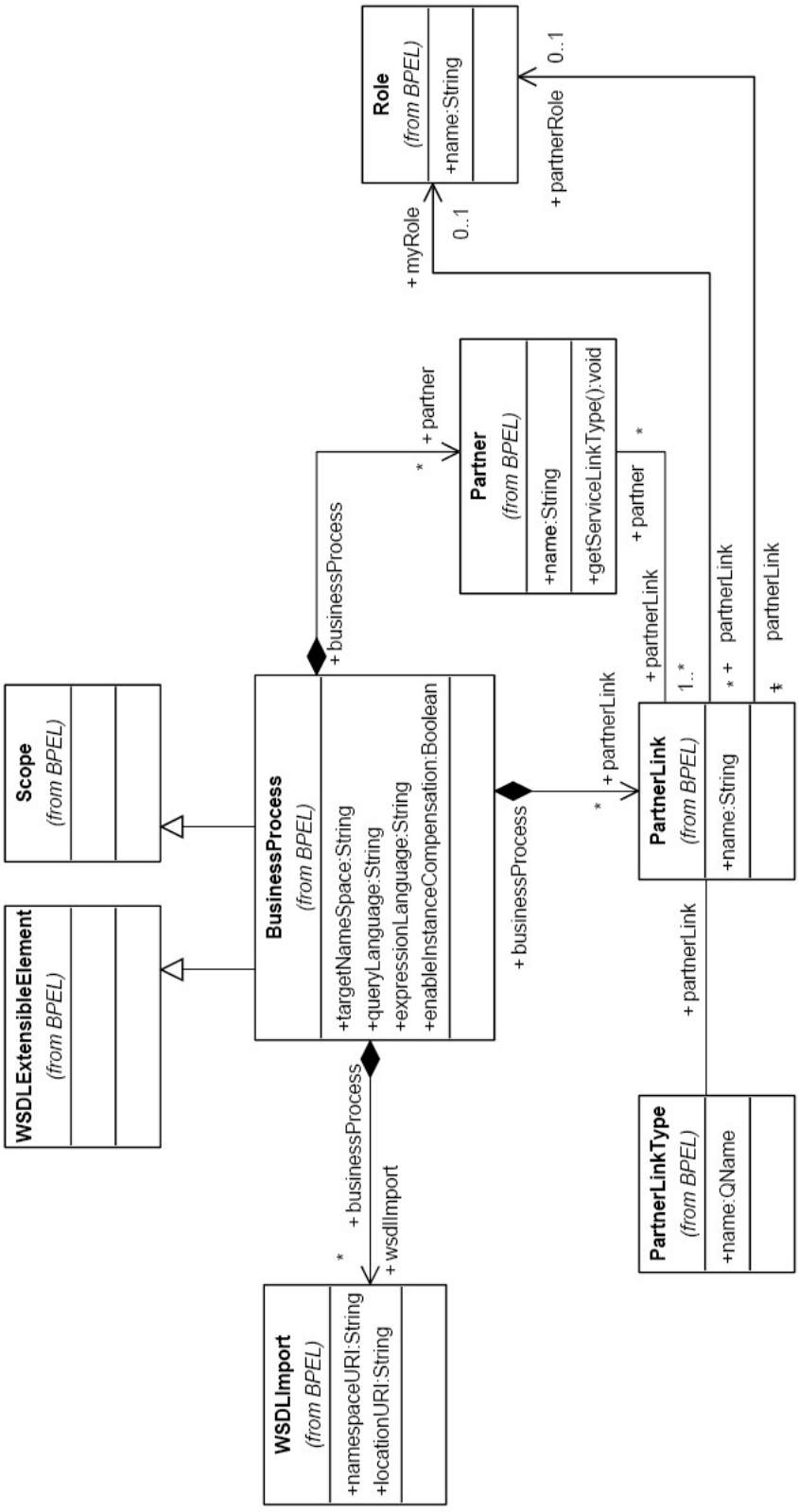

Figure 2.1: Business Process

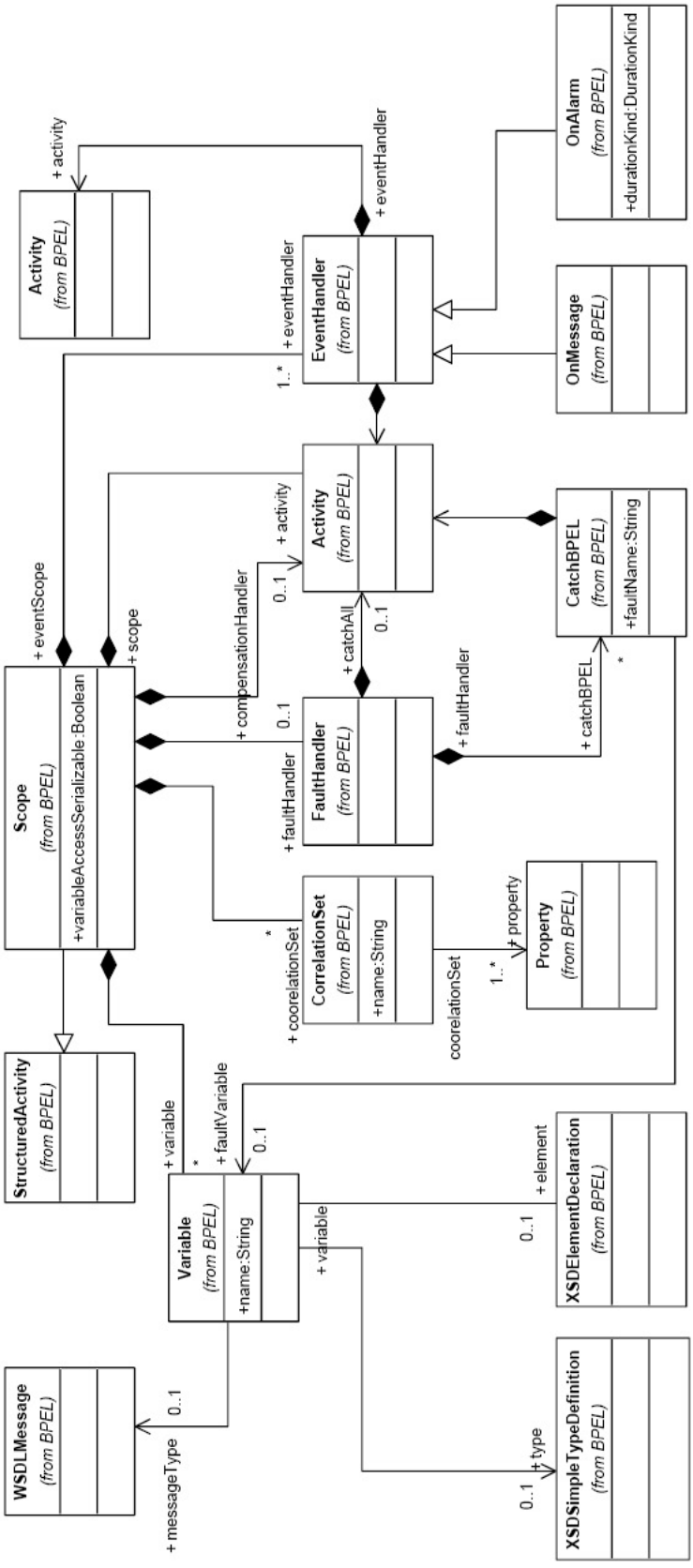

Figure 2.2: Scope, Variable and CorrelationSet

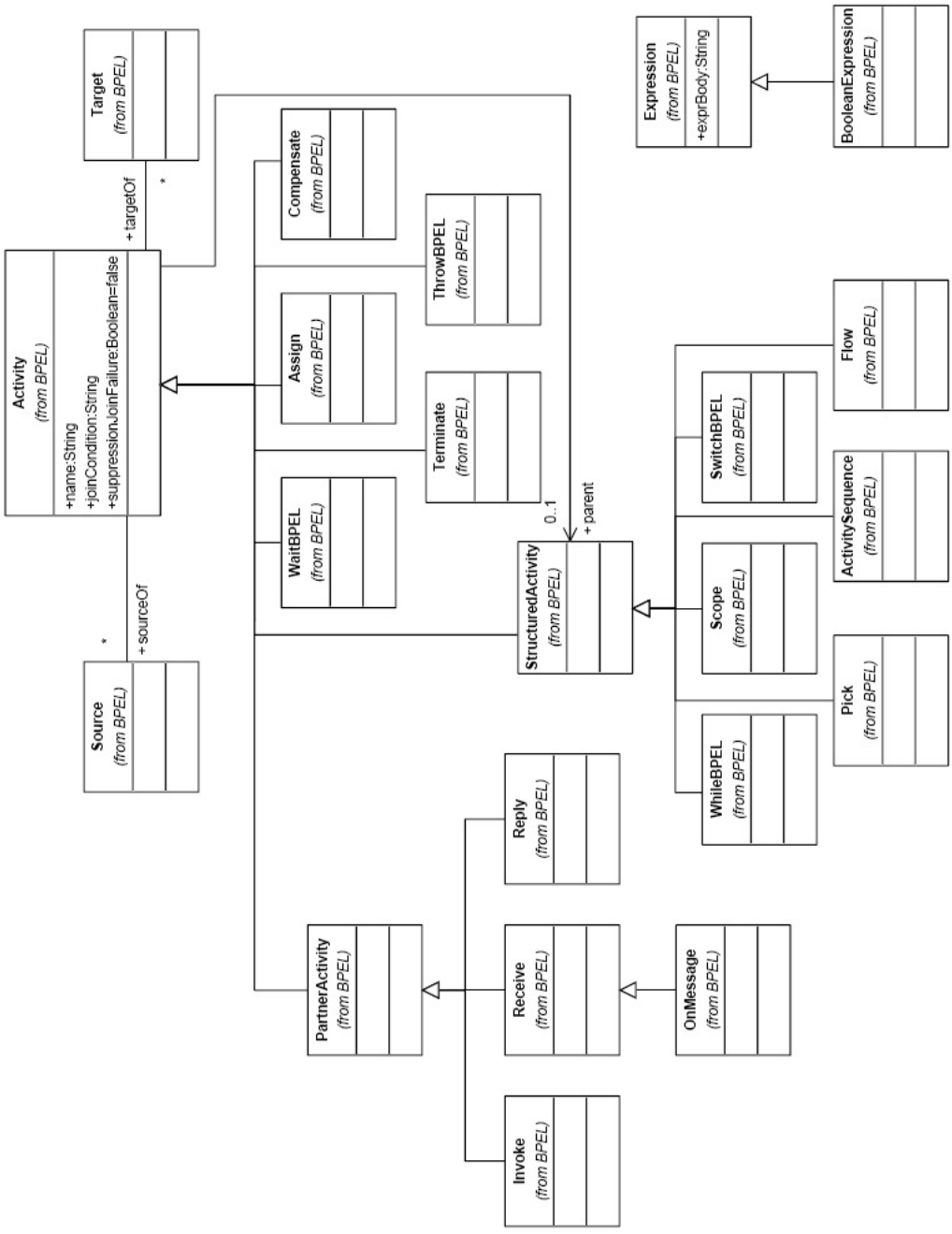

Figure 2.3: Activity Hierarchy

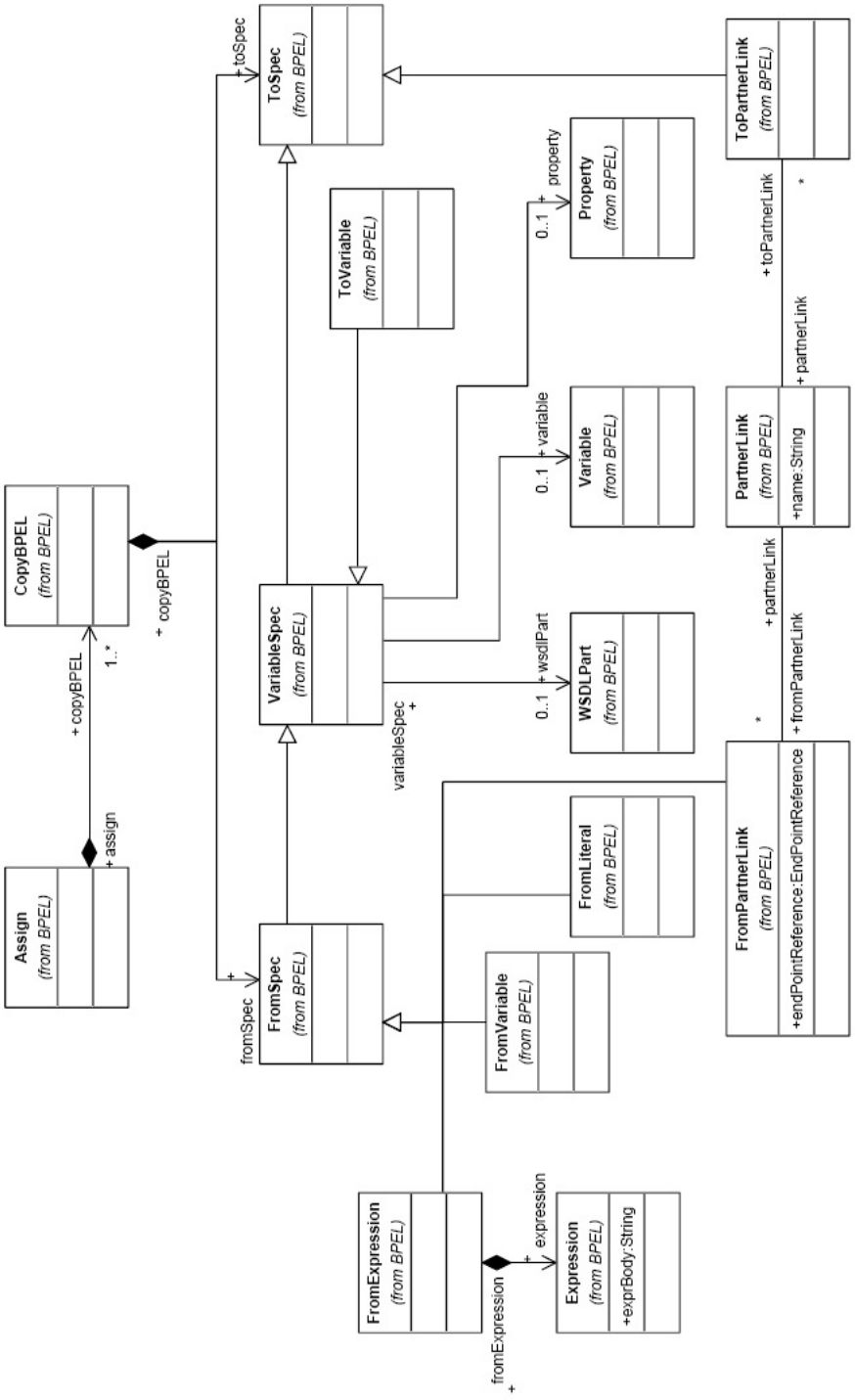

Figure 2.4: Assign Activity

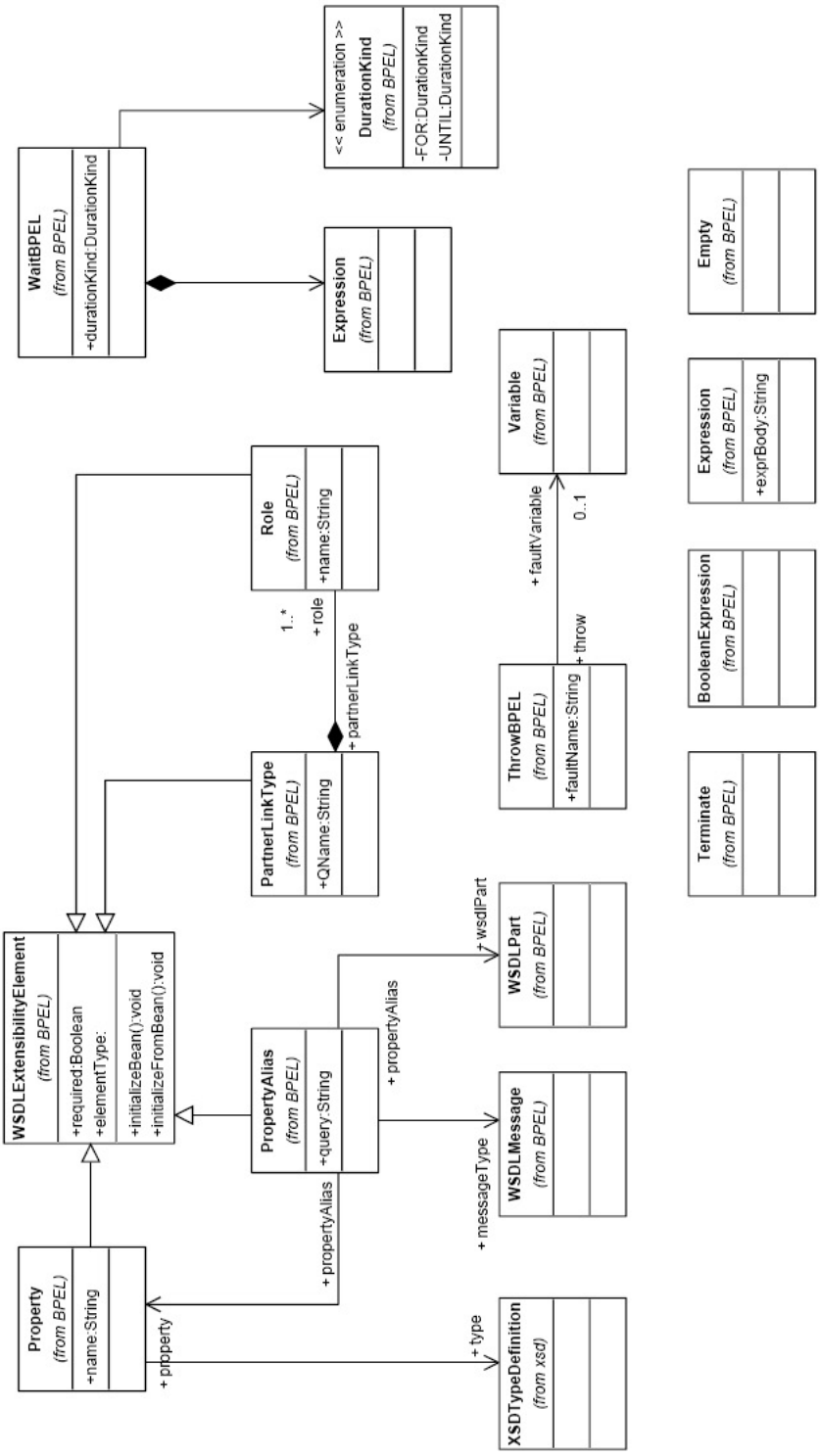

Figure 2.5: WSDL Extensions, Basic Activities

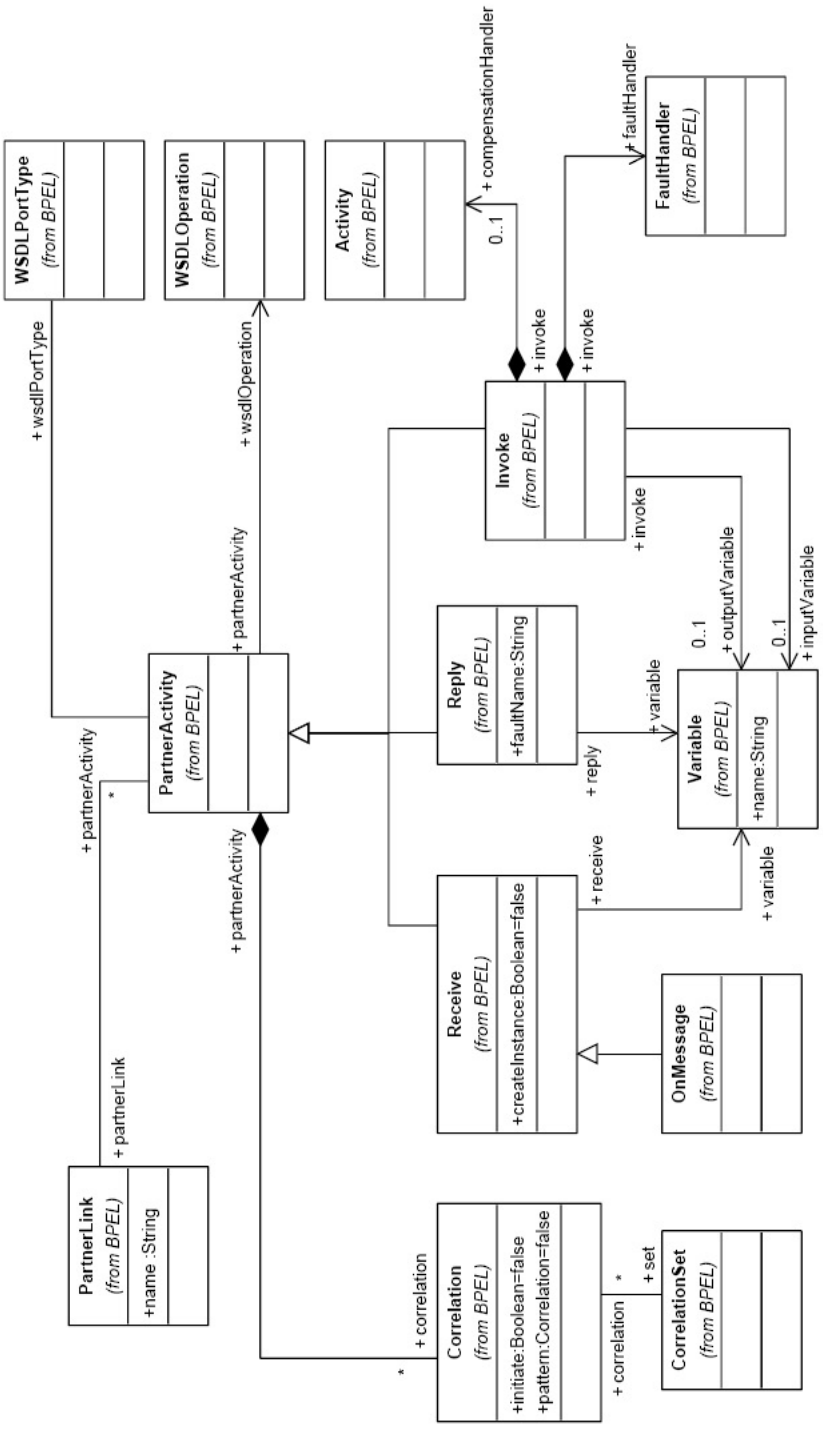

Figure 2.6: Partner Activities

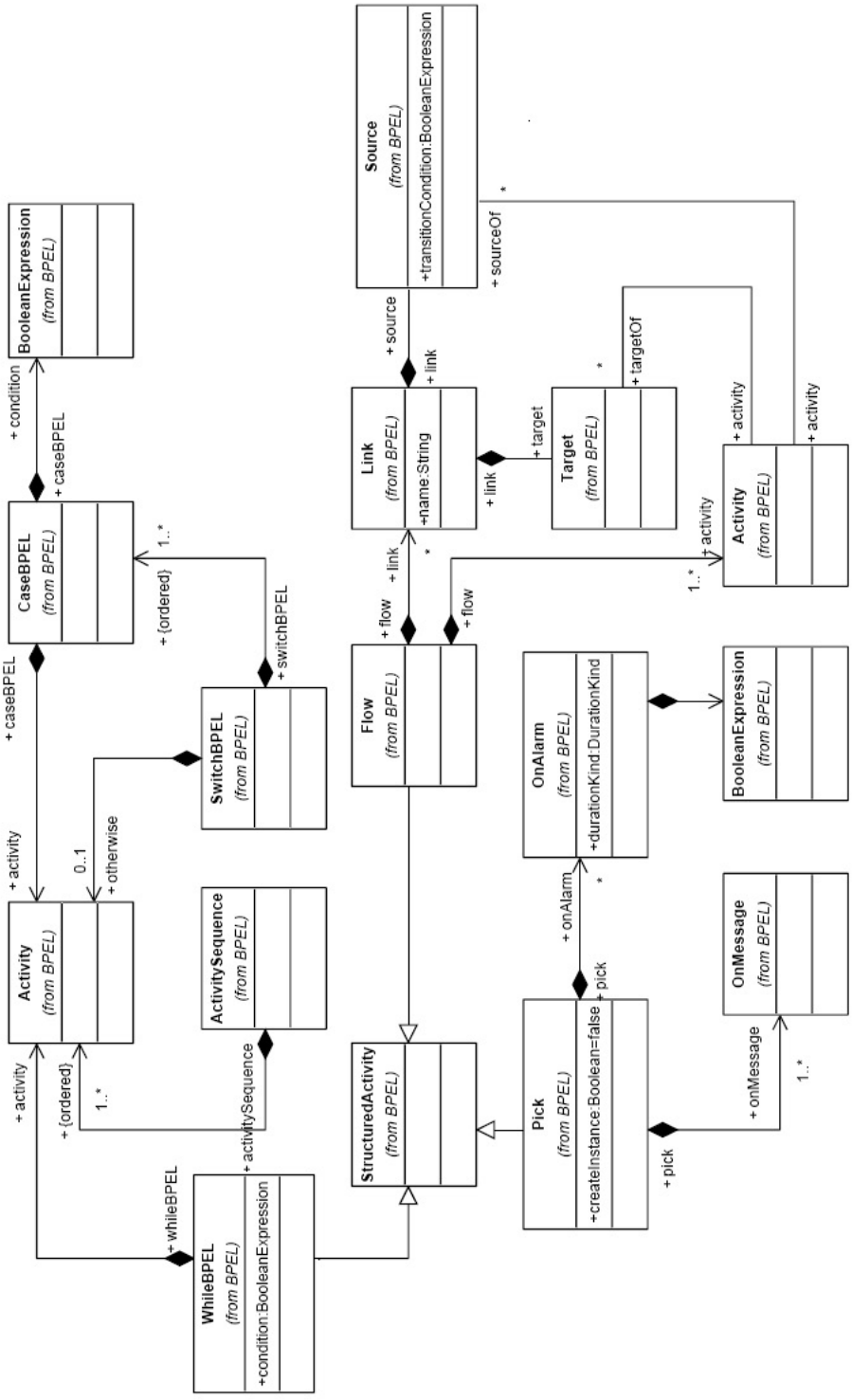

Figure 2.7: Structured Activities

Chapter 3

Need for Constraints

3.1 Introduction to OCL

Object Constraint Language (OCL) [5] is a language that enables one to describe expressions and constraints on object-oriented models and other object modeling artifacts. An expression is an indication or specification of a value. It is a restriction on one or more values of (part of) an object-oriented model or system.

- OCL is a standard query language, which is part of the Unified Modeling Language (UML) set by the Object Management Group (OMG) as a standard for object-oriented analysis and design.
- OCL is a pure expression language. Therefore, an OCL expression is guaranteed to be without side effect; it cannot change anything in the model. This means that the state of the system will never change because of an OCL expression, even though an OCL expression can be used to specify a state change, e.g. in a post-condition. All values for all objects, including all links, will not change. Whenever an OCL expression is evaluated, it simply delivers a value.
- OCL is not a programming language, so it is not possible to write program logic or flow control in OCL. We cannot invoke processes or activate non-query operations within OCL.
- OCL is a modeling language in the first place, not everything in it is promised to be directly executable.
- OCL is a typed language, so each OCL expression has a type. In a correct OCL expression all types used must be type conformant. For example, we cannot compare an Integer with a String. Types within OCL can be any kind of Classifier within UML.
- As a modeling language, all implementation issues are out of scope and cannot be expressed in OCL. Each OCL expression is conceptually atomic. The state of the objects in the system cannot change during evaluation.

3.2 Types of Constraints

There are four types of constraints:

- An **invariant** is a constraint that states a condition that must always be met by all instances of the class, type, or interface. An invariant is described using an expression that evaluates to true if the invariant is met. Invariants must be true all the time.
- A **precondition** to an operation is a restriction that must be true at the moment that the operation is going to be executed. The obligations are specified by post-conditions.
- A **postcondition** to an operation is a restriction that must be true at the moment that the operation has just ended its execution.
- A **guard** is a constraint that must be true before a state transition fires.

3.3 Why OCL?

Being 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 to the precise specification of tag names and attributes, the BPEL standard defines approximately 20 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 based BPEL document, which in actual fact violates one or more usage constraints.

OCL language is initially based primarily on Set theory concepts. It is defined initially as a “constraint” language with its core expression part can be used as an object-based Query Language. OCL is used in this project to specify the well-formedness rules of the BPEL metamodel. Each well-formedness rule is given in the form of an OCL expression, which is an invariant for the involved class.

3.4 Constraints for BPEL [1]

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, which is not captured by XML structure of a BPEL document. OCL, through the use of the “def” context, enables us to give additional model properties, which can subsequently be used to specify the required OCL expressions.

The quoted textual constraints are translated into OCL constraints placed in the context of classes from the BPEL metamodel.

Each subsection corresponds to a set of constraints taken from one subsection of the standard document.

3.4.1 Business Process

"...
< partnerLinks >?
<!– Note: At least one role must be specified. –>"

```
context PartnerLink
 inv atLeastOneRoleMustBeDefined :
  not ( self.myRole.oclIsUndefined() and
    self.partnerRole.oclIsUndefined() )
```

"...
< faultHandlers >?
<!– Note: There must be at least one fault handler or default. –>"

```
context FaultHandler
 inv atLeastOneFaultHandlerOrDefault :
  self.catchAll.oclIsUndefined() implies self.catchBPEL->notEmpty()
```

"...
< eventHandlers >?
<!– Note: There must be at least one onMessage or onAlarm handler. –>"

```
context Scope
 inv atLeastOneOnMessageOrOnAlarmHandler :
   self.eventHandler->notEmpty() implies
   self.eventHandler->notEmpty()
```

The BPEL metamodel defines a Business Process to be a subtype of Scope in order to reuse the structure of the Scope element. However, a Business Process is not mentioned in the BPEL specification to be a subActivity of any other activity, thus we place an additional constraint as follows:

```
context BusinessProcess
 inv processIsNotASubActivity :
   self.parent.oclIsUndefined()
```

3.4.2 Partner Definitions must not overlap [2]

The following is the constraint for restriction on connections between a "partner" construct and a "partnerlink" construct.

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

The two constructs as in Figure 3.1 are represented in BPEL as sub elements of the top level "BusinessProcess" element. Considering them which shows a valid and an invalid process specification: 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.

With enforcing this constraint, we model the relationship between Partner and PartnerLink as an [0..1]-to-[0..*] association, this states that any one partnerLink can only be

```
<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>
```

Figure 3.1: Invalid and Valid XML Segment

associated to a single partner and thus the above constraint would not be violated. However, if we consider the generating a BPEL validator, 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.

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.

We have adopted the second approach and the necessary OCL constraint is given below.

```
context BusinessProcess
  inv partnerDefinitionsMustNotOverlap :
    self.partner.partnerLink->asSet()->asBag()
      = self.partner.partnerLink->asBag()
```

With regard to 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.

3.4.3 Variable Options

"...
The messageType, type or element attributes are used to specify the type of a variable. Exactly one of these attributes must be used."

```
context Variable
  inv variableRefersToOneItem :
    Bag{ not self.messageType.oclIsUndefined(),
      not self.type.oclIsUndefined(),
      not self.element.oclIsUndefined() }->count(true) = 1
```

3.4.4 Source and Target of Activities

"...An activity MAY declare itself to be the source of one or more links by including one <source> elements. Each <source> element MUST use a distinct link name. Similarly, an MAY declare itself to be the target of one or more links by including one or more elements. Each <source> element associated with a given activity MUST use a link name from all other <source> elements at that activity. Each <target> element associated with activity MUST use a link name distinct from all other <target> elements at that activity."

```
context Activity
  inv eachSourceElementMustUseDistinctLinkName :
    self.sourceOf.link->isUnique(s|s.name)
  inv eachTargetElementMustUseDistinctLinkName :
    self.targetOf.link->isUnique(s|s.name)
```

3.4.5 Pick

"...A special form of pick is used when the creation of an instance of the business process could occur as a result of receiving one of a set of possible messages. In this case, the pick itself has a createInstance attribute with a value of yes (the default value of the attribute is no). In such a case, the events in the pick must all be inbound messages and each of those is equivalent to a receive with the attribute "createInstance=yes". No alarms are permitted for this special case."

```
context Pick
 inv createInstancePickImpliesAllEventsAreCreateInstanceReceives :
  self.createInstance
 implies
  self.onMessage->forAll( act | act.createInstance )
 and
  self.onAlarm->isEmpty()
```

"... Each pick activity MUST include at least one onMessage event."
This constraint is imposed by the 1..* multiplicity on the Pick-OnMessage association. However we add an invariant to check it.

```
context Pick
 inv pick_onMessage_MultiplicityAtLeastOne :
  self.onMessage->size() >= 1
```

3.4.6 Flows and Links

"... A link has a name and all the links of a flow activity MUST be defined separately within the flow activity."

This constraint is imposed by the Model. Links are contained by a Flow (and can't be included anywhere else). Links only have two ends, therefore each link is separate.

"... The source of the link MUST specify a source element specifying the link's name and the target of the link MUST specify a target element specifying the link's name."

This constraint is imposed by the Model. Within a Link, Source and Target elements are not optional.

"... 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."

To express this constraint in OCL, we require the method **subActivities** to be defined for each subtype of Activity. The method returns a set containing all activities directly nested within that Activity. Also required is a method **allSubActivities** which returns all nested and sub-nested activities. For basic Activities this set will typically be empty.

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

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

"... In addition, a link that crosses a fault-handler boundary MUST be outbound, that is, it MUST have its source activity within the fault handler and its target activity within a scope that encloses the scope associated with the fault handler."

```
context FaultHandler
 inv boundryCrossing :
  let allSubActivities : Set(Activity) =
   Set { self.catchAll }
    ->union( self.catchBPEL.activity->asSet() ).allSubActivities->asSet()
  in
    allSubActivities->includesAll( allSubActivities.sourceOf.activity )
  and
    allSubActivities.sourceOf.link.target.activity->forAll( tgtAct |
     self.scope.allParents->exists( act |
     act.allSubActivities->includes(tgtAct)
     )
    )
```

Chapter 4

Java Architecture for XML Binding (JAXB)

4.1 What is JAXB?

XML data binding relieves the pain of any Java programmer who has ever winced at having to work with a document-centric processing model. Unlike SAX and DOM, which forces to think in terms of a document's structure, XML data binding lets us think in terms of the objects the structure represents. It does so by realizing the structure as a collection of Java classes and interfaces.

With JAXB, Java Architecture for XML binding [9], we can generate Java classes from XML schemas by means of a JAXB binding compiler. The binding compiler takes XML schemas as input, and then generates a package of Java classes and interfaces that reflect the rules defined in the source schema. These generated classes and interfaces are in turn compiled and combined with a set of common JAXB utility packages to provide a binding framework. But our interests are restricted within the usage of unmarshalling the BPEL schema document and accessing its content objects to read a XML document.

The JAXB binding framework provides methods for unmarshalling XML instance documents into Java content trees – a hierarchy of Java data objects that represent the source XML data – and for marshalling Java content trees back into XML instance documents. The JAXB binding framework also provides methods for validating XML content as it is unmarshalled and marshalled. Moreover, Java developers do not need to be well-versed in the intricacies of SAX or DOM processing models, or even in the arcane language of XML schema, to take advantage of ubiquitous, platform-neutral XML technologies.

Now developers have new version of Java Architecture for XML Binding [7] (JAXB 2.0) at their disposal that can make it easier to access XML documents.

4.2 Comparison with JAXB 1.0

- JAXB 1.0 creats a copy of attributes so that it can be processed later. 2.0 no longer does this.
- JAXB 2.0 defers the computation of raw element name until necessary, and most commonly just avoids it altogether.

- State machine model is much simplified in 2.0, resulting in smaller code size, smaller memory footprint, and smaller number of method invocations. This is of greater advantage in our case since large number of classes generated for a BPEL schema.

4.3 How Does JAXB fit in our case?

A JAXB implementation comprises eight core components:

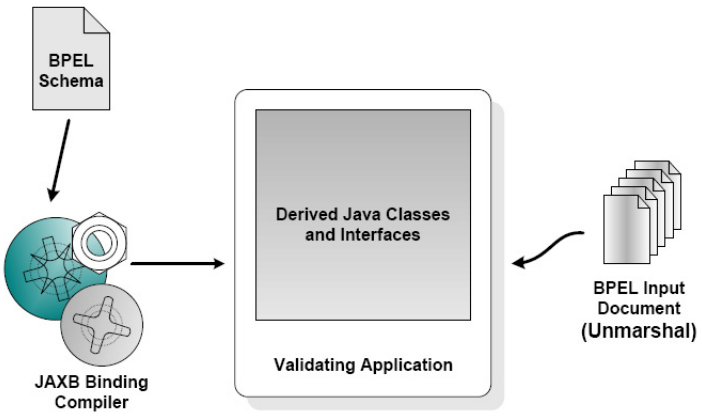

Figure 4.1: General JAXB Overview

4.3.1 BPEL Schema

BPEL schema uses XML syntax to describe the relationships among elements, attributes and entities in a BPEL document. An XML document that conforms to a particular schema is referred to as an instance document.

4.3.2 Binding Declarations

By default, the JAXB binding compiler binds Java classes and packages to a source XML schema based on rules defined in the JAXB Specification. In most cases, the default binding rules are sufficient to generate a robust set of schema-derived classes from a wide range of schemas. There may be times, however, when the default binding rules are not sufficient for our needs. JAXB supports customizations and overrides to the default binding rules by means of binding declarations made either inline as annotations in a source schema, or as statements in an external binding customization file that is passed to the JAXB binding compiler.

4.3.3 Binding Compiler

The JAXB binding compiler is the core of the JAXB processing model. Its function is to transform, or bind, a source XML schema to a set of JAXB content classes in the Java programming language. Basically, we run the JAXB binding compiler using a BPEL schema (optionally with custom binding declarations) as input in our case, and the binding compiler generates Java classes that map to constraints in the source BPEL schema.

4.3.4 Binding Framework Implementation

The JAXB binding framework implementation is a runtime API that provides interfaces for unmarshalling, marshalling, and validating XML content in a Java application. The binding framework comprises interfaces in the **javax.xml.bind** package.

4.3.5 Schema-Derived Classes

These are the schema-derived classes generated by the binding JAXB compiler. The specific classes will vary depending on the input schema.

4.3.6 BPEL Validation Application

In the context of JAXB, a Java application is a client application that uses the JAXB binding framework to unmarshal XML data, validate and modify Java content objects, and marshal Java content back to XML data. We use these content objects to access the BPEL contents and map them with the OCTOPUS Constraint Code which is discussed in the next chapter.

4.3.7 BPEL Input Documents

BPEL documents, we wish to validate would be given as input to the above Validation Application. Detailed description is provided in the next section. In JAXB, the unmarshalling process supports validation of the XML input document against the constraints defined in the source schema. This validation process is optional, however, and there may be cases in which one should know by other means that an input document is valid and so validation could be skipped for performance reasons. In any case, validation before or during unmarshalling is important, because it assures that an XML document generated during marshalling will also be valid with respect to the source schema. But we do not validate using the default validation process during unmarshalling.

4.3.8 XML Output Documents

In JAXB, marshalling involves parsing an XML content object tree and writing out an XML document that is an accurate representation of the original XML document, and is valid with respect the source schema. But marshalling is of no significance for us in the present venture of the project.

4.4 JAXB 2.0 Binding Process for BPEL

There general steps in the JAXB data binding processfor BPEL would be:

4.4.1 Generate Classes

A BPEL schema is used as input to the JAXB binding compiler to generate source code for JAXB classes based on the required schema. With the use of binding compiler we specify the context path for packages that are to be generated for the specific BPEL schema. In our case, we specify the context path to be **org.xmlsoap.schemas.wsdl.business_process**.

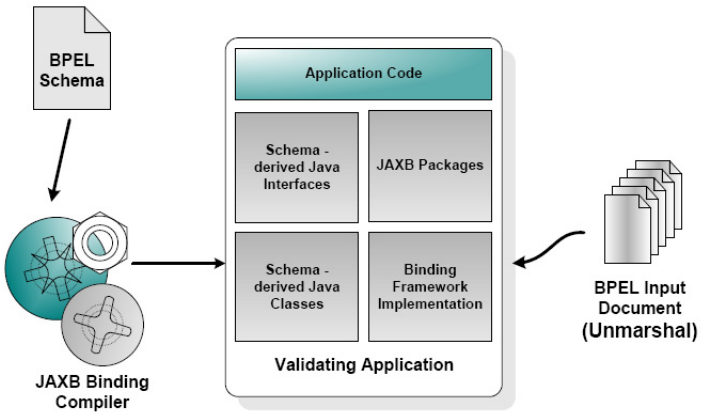

Figure 4.2: Core JAXB Components

4.4.2 Compile Classes

All of the generated classes, source files, and application code must be compiled.

4.4.3 Unmarshal

BPEL documents written according to the constraints in the BPEL schema are unmarshalled by the JAXB binding framework. Unmarshalling does creating a tree of content objects that represents the content and organization of the BPEL schema document. To unmarshall it the following steps are followed.

Create a JAXB Context Object

This object provides the entry point to the JAXB API. When an object is created, we need to specify a context path. This is a list of one or more package names that contain interfaces generated by the binding compiler.

For example, the following code snippet creates a JAXBContext object whose context path is **org.xmlsoap.schemas.wsdl.business_process**, the package that contains the interfaces generated for the BPEL schema:

```
import org.xmlsoap.schemas.wsdl.business_process.*;
public class Main {
    // This Application demonstrates how to unmarshal a BPEL document
    // into a Java content tree and access data contained within it.
    public static void main( String[] args ) {
        try {
            JAXBContext jc = JAXBContext.newInstance(
                    "org.xmlsoap.schemas.wsdl.business_process");
            // create an Unmarshaller
            Unmarshaller u = jc.createUnmarshaller();
```

Figure 4.3: JAXB Content Object

Create an Unmarshaller object

This object controls the process of unmarshalling. In particular, it contains methods that perform the actual unmarshalling operation. For example, the code snippet in Figure 4.3 creates an Unmarshaller object after creating JAXBContext object.

Call the unmarshal method

This method does the actual unmarshalling of the BPEL document. For example, the following code snippet unmarshals the data in the **AmazonFlow.bpel** file: The **topElem**

```
JAXBElement<?> topElem = (JAXBElement<?>) unmarshal
        .unmarshal(new FileInputStream("AmazonFlow.bpel"));
```

refers to the top node of object graph generated for **AmazonFlow.bpel**. With the classes that JAXB compiler generated for the schema, we could obtain the data for each type of element and attribute with the generated get and set methods.

JAXB 2.0 extensively uses generics, a new introduction with J2SE 5.0 which results in clean generated code, better user readability and robustness. More information about generics is given in appendix B.

Generate Content Tree

The unmarshalling process generates a content tree of data objects instantiated from the generated JAXB classes; this content tree represents the structure and content of the source BPEL documents. The Figure 4.4 refers to the content tree generated for **AmazonFlow.bpel**.

Validate

The unmarshalling process optionally involves validation of the source XML documents before generating the content tree. We do not validate the BPEL documents with the optional JAXB validation feature.

Validation could be done before unmarshalling the documents into content tree and also to validate the changes made to the BPEL document after marshalling its content tree back to a BPEL document. But this would the future extension for the project.

Figure 4.4: Content Tree for AmazonFlow.bpel

Chapter 5

Validation

5.1 Performing a BPEL Validation with OCL

A set of UML Class Diagrams have been created to model the structural aspects of the BPEL language. We use Poseidon to draw the Class Diagrams and export them as XMI. When imported into an Eclipse project with Octopus Nature enabled, the Uml Diagrams would be converted into Octopus representation for UML.

An Eclipse Octopus project [6] would have two special folders.

- The *"folder for UML model"*, in short the ***"model folder"***, is the folder where the files that specify the BPEL metamodel that are used when writing OCL expressions, are stored.
- The *"folder for OCL expressions"*, in short the ***"expressions folder"***, is the folder where the files that contain OCL expressions are stored.

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. Firstly, with reference to the Figure 5.1 we use Octopus to generate the code for the BPEL metamodel and the Constraints listed in the previous chapter.Errors in constraints against the metamodels would be detected at this phase itself. After having the corrected BPEL metamodel with the ocl constraints. we generate the code using Octopus OCL tool.

Initially, most of the classes as per specified in the research paper would violate the Java reserved word group and hence such classes have been renamed with BPEL being appended to their existing names. For Example, SwitchBPEL, CaseBPEL.. etc.

Secondly, JAXB compiler is used to generate classes and interfaces as shown in chapter .. with BPEL schema as input to access any data of any BPEL file provided as input in the validation program. Thus using these two tools (Octopus OCL and JAXB) the core code for validating a BPEL can be generated.

There are several issues worth drawing out regarding our experiences of mapping the 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.

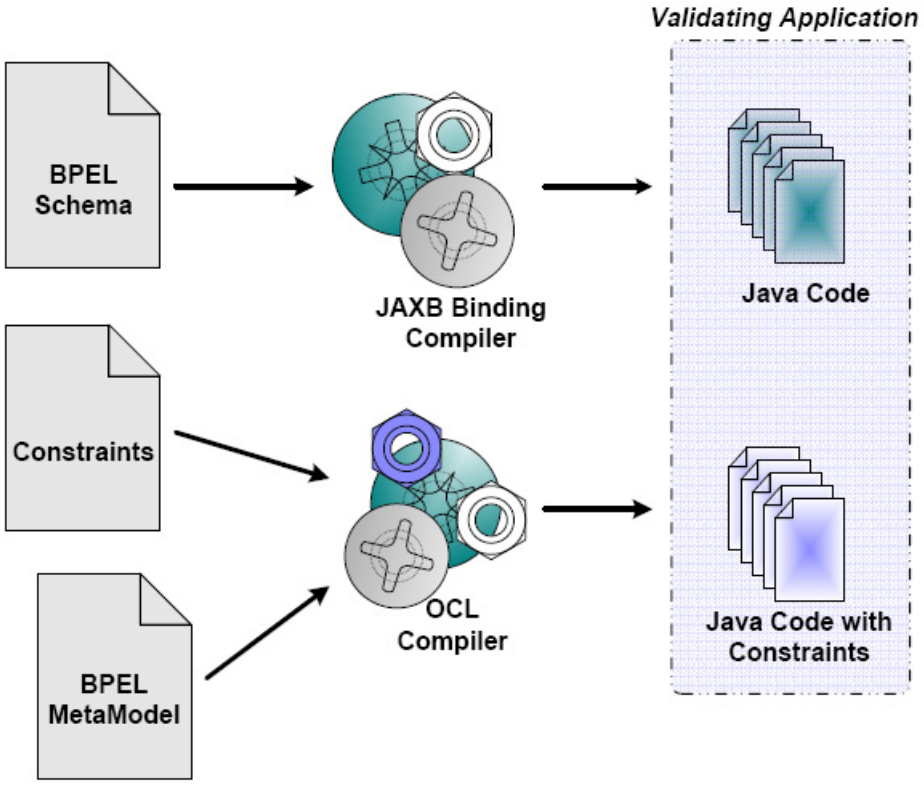

Figure 5.1: Code Generation from Schema, Metamodel and Constraints

The first issue we consider mentioning is the mapping of two or more package represention in the Uml for BPEL. Consider the UML snippet from a typical XMI imported UML file representing BPEL schema,

```
<package> BPEL
  + <class> Property <specializes> wsdl::ExtensibilityElement
   <attributes>
    + name: String;
  <endclass>
<endpackage> //BPEL

<package> wsdl
  +<class> ExtensibilyElement
  ..................
  ..................
<endpackage>//wsdl
```

The Snippet shows two packages. named BPEL, WSDL. Generation fails to generate correct code representing the inheritance between the Property Class of BPEL with that of ExtensiblityElement Class of WSDL.

So we made changes in such that all classes are placed in the same package and the classes of both WSDL package and XSD package are identified with the package names appended to their class names.

The second issue, would be the presence of multiple inheritance in BPEL metamodel. Preventive Steps were taken in such a way the classes inherit not more than one class.

After having the classes generated with JAXB and Octopus, we have to map the instances of JAXB towards the Constraint code generated by Octopus. Octopus has an unconfigurable Schema which it uses to validate the Xml files for reading and writing back into them. This drawback forces the implementation of JAXB for mapping its instances with constraints required for validation.

After unmarshalling the file to be validated with JAXB, we obtain an object graph of all elements available in it. We implement visitor pattern to traverse through the object graph of JAXB generated classes. As visiting these classes we instantiate their respective classes (with OCL Constraint). With reference to the code snippet shown,

```
TProcess tPro = (TProcess) topElem.getValue();
BpelJaxbAstWalker walker = new BpelJaxbAstWalker();
BpelJaxbToString visitor = new BpelJaxbToString();
walker.accept(tPro, visitor);
```

The method **accept()** of **Class BpelJaxbAstWalker** implements the visitor pattern. For Example, In Case of different types of Activities available for BPEL, the Figure 5.2 shows the hierarchy of types of activities. With reference to this example,

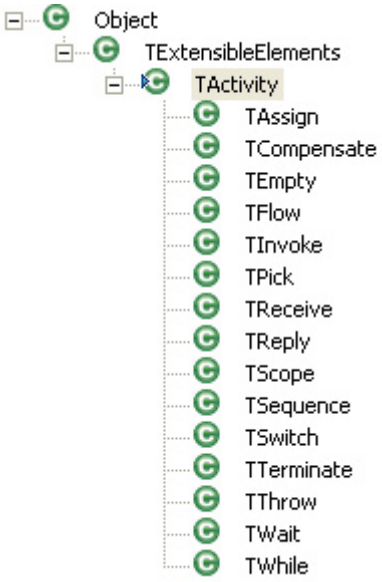

Figure 5.2: Hierarchy of Activity

At any point of time when the traversing for Activity has to be done, then a walk on all the types of Activity would be done comparing whether the object is an instance of any of the activity's subclass types.

```
private Object visitTActivity(TActivity exp,
    IBpelJaxbAstVisitor visitor)
{
Object result = null;
if (exp instanceof TAssign) {
    result = visitTAssign((TAssign) exp, visitor);
} else
      ..................
      ..................
}
```

When a match is found a step down is done by the walker to visit the Activity's subclass. Let us consider that the **walker** visits **Class TAssign**, then an instance of the OCL-constrained **Class Assign** is instantiated and added to a map with the JAXB object for **TAssign** as the key and newly instanciated Assign as the value pair.

```
 private Map instanceMap = new HashMap();
 ................
 ................
private void visitTAssign(TAssign exp, IBpelJaxbAstVisitor visitor)
{
if (instanceMap.get(exp) != null)
return; // we've already visited this element
AssignGEN  inst = new AssignGEN();
    instanceMap.put(exp,inst);
    ..............
 }
```

This step is followed for all elements of the BPEL file. But in general, we have to follow a structure in visiting the nodes and this could be only done with the vistior pattern.

After creating all the instances required, it is required to map the values of BPEL file that is accessible with JAXB to that of methods of type **setXXX()** in Constraint Classes.

This could be done with the aid of previously created Hash map which has objects of JAXB with that of OCL objects as Key - Value pairs. As the objects are visited the **getXXX()** methods of JAXB are mapped to that of the **setXXX()** methods of appropriate OCL classes.

After the completion of mapping, the invariants are checked against the instances by calling **checkMultiplicities()** and **checkAllInvariants()** while traversing the object graph using the visitor pattern. These methods could be made to print the error messages directly to the console or to a GUI.

Chapter 6

Conclusion

In this paper, the use of OCL and UML to provide a precise version of natural language constraints on the structuring of BPEL XML documents is shown. 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 validation is particularly useful in the case of BPEL, as some of the natural language constraints are ambiguous or complex to understand.

Additionally, 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 are being discussed. Few language constraints could not be modeled with OCL for the given BPEL metamodel. With these constraints as a part of Constraint code would have made the validation process even more powerful.

Chapters of importance for this project could be narrowed with the chapter 4 discussing about customizing JAXB to generate classes for a BPEL Schema and chapter 5 highlighting the implementation of visitor pattern for mapping instances of JAXB to that of Octopus generated Constraint Code.

As a future improvement for this project, a Graphical User Interface could be developed which could have possibility of importing BPEL XML document, editing the instances of BPEL and marshalling back to the document. A dedicated button could be provided for validating constraints against the instance of an input BPEL file.

Finally the paper has discussed issues regarding the automatic generation of code, in this case a BPEL validator, using OCL constraints as the source and steps to be taken in mapping the JAXB instances to that of Octopus Constraint Code.

Chapter 7

Appendix A

7.1 Defined Properties and Operations - OCL Constraints

7.1.1 Define Property - subActivities : Set

```
context Activity::subActivities : Set(Activity)
derive: Set {}

context ActivitySequence::subActivities : Set(Activity)
derive: self.activity->asSet()

context Pick::subActivities : Set(Activity)
derive: self.onAlarm.activity->union( self.onMessage )->asSet()

context SwitchBPEL::subActivities : Set(Activity)
derive: self.caseBPEL.activity->asSet()->union(
          Set{self.otherwise} )

context Scope::subActivities : Set(Activity)
derive:
  let
    compHndlr : Set(Activity) = Set{self.compensationHandler},
    fltHndlr : Set(Activity) =
      Set { self.faultHandler.catchAll }
       ->union( self.faultHandler.catchBPEL.activity->asSet() ),
      evntHndlr : Set(Activity) =
      Set { self.eventHandler.activity }->flatten()
  in
    Set{self.activity}
    ->union(compHndlr)
    ->union(fltHndlr)
    ->union(evntHndlr)
```

7.1.2 Define Property - allSubActivities : Set

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

7.1.3 Definition of Property - initialActivities : Set

This property is intended to return a Set containing all possible activities that could occur if this activity is expected to occur.

Basic Activities

```
context Activity::initialActivities : Set(Activity)
derive: Set{self}
```

Structured Activities

```
context WhileBPEL::initialActivities : Set(Activity)
derive: self.activity.initialActivities

context Flow::initialActivities : Set(Activity)
derive:
  self.activity->select( a |
    a.targetOf->isEmpty()
  ).initialActivities->asSet()

context Pick::initialActivities : Set(Activity)
derive:
  self.onAlarm.activity.initialActivities
    ->union( self.onMessage.initialActivities )
    ->asSet()

context SwitchBPEL::initialActivities : Set(Activity)
derive:
  let
    otherW : Set(Activity) = if self.otherwise.oclIsUndefined()
      then
      Set{}
    else
      self.otherwise.initialActivities
    endif
  in
  self.caseBPEL.activity.initialActivities->asSet()
    ->union( otherW )->asSet()

context Scope::initialActivities : Set(Activity)
derive: self.activity.initialActivities
```

7.1.4 Definition of Property - allBasicActivities : Set

Basic Activities

```
context Activity::allBasicActivities : Set(Activity)
derive: Set{self}
```

Partner Activities

– OnMessage is a subtype of Receive

```
context Receive::allBasicActivities : Set(Activity)
derive: Set{self}

context Reply::allBasicActivities : Set(Activity)
derive: Set{self}

context Invoke::allBasicActivities : Set(Activity)
derive: Set{self}
```

Structured Activities

```
context WhileBPEL::allBasicActivities : Set(Activity)
derive: self.activity.allBasicActivities->asSet()

context Flow::allBasicActivities : Set(Activity)
derive: self.activity.allBasicActivities->asSet()

context ActivitySequence::allBasicActivities : Set(Activity)
derive: self.activity.allBasicActivities->asSet()

context Pick::allBasicActivities : Set(Activity)
derive:  self.onAlarm.activity.allBasicActivities
  ->union( self.onMessage.allBasicActivities )->asSet()

context SwitchBPEL::allBasicActivities : Set(Activity)
derive:
  let otherW : Set(Activity) = if self.otherwise.oclIsUndefined()
    then
    Set{}
  else
    self.otherwise.allBasicActivities
  endif
  in
    self.caseBPEL.activity.allBasicActivities->asSet()
      ->union( otherW )->asSet()

context Scope::allBasicActivities : Set(Activity)
derive: self.activity.allBasicActivities
```

7.1.5 Definition of Property - next : Set

Property prev is similarly defined; however the definition is not explicitly given.

Basic and Partner Activities

```
context Activity::next(prev: Activity) : Set(Activity)
body:
  self.parent.next(self)->collect( n |
    if n.oclIsTypeOf(StructuredActivity) then
      n.initialActivities
    else
      Set{n}
    endif
  )->flatten()->asSet()
```

7.1.6 Definition of Property - orderedSubActivities : Set

Note: activities in compensation, fault and event handlers are treated as subActivities.

Basic Activities

```
context Activity::orderedSubActivities : Sequence(Activity)
derive: Sequence {}
```

Partner Activities

```
context OnMessage::subActivities : Set(Activity)
derive: Set { self.activity }

context Invoke::subActivities : Set(Activity)
derive:
  let
    compHndlr : Set(Activity) = Set{self.compensationHandler},
    fltHndlr : Set(Activity) =
      Set { self.faultHandler.catchAll }
      ->union( self.faultHandler.catchBPEL.activity->asSet() )
  in
    fltHndlr->union( compHndlr )
```

Structured Activities

```
context WhileBPEL::orderedSubActivities : Sequence(Activity)
derive: Sequence { self.activity }

context Flow::orderedSubActivities : Sequence(Activity)
derive: self.activity->asSequence()

context ActivitySequence::orderedSubActivities :
 Sequence(Activity)
derive: self.activity->asSequence()
```

```
context Pick::orderedSubActivities : Sequence(Activity)
derive:
  self.onAlarm.activity->asSequence()
    ->union( self.onMessage->asSequence() )

context SwitchBPEL::orderedSubActivities : Sequence(Activity)
derive:
  self.caseBPEL->collectNested(c|c.activity)->including
  ( self.otherwise )
    ->asSequence()

context Scope::orderedSubActivities : Sequence(Activity)
derive:
  let
   -- undefined's are not put into sets, so these are empty
   -- if navigations are undefined
    compHndlr : Sequence(Activity) =
    Sequence { self.compensationHandler },
    fltHndlr : Sequence(Activity) =
    Sequence { self.faultHandler.catchAll }
      ->union( self.faultHandler.catchBPEL.activity
      ->asSequence()),
    evntHndlr : Sequence(Activity) =
    self.eventHandler.activity->asSequence()
  in
    Sequence{self.activity}
      ->union(compHndlr)
      ->union(fltHndlr)
      ->union(evntHndlr)
```

7.1.7 Definition of Property - allOrderedSubActivities : Set

```
context Activity
def: allOrderedSubActivities : Sequence(Activity) =
  if self.oclIsKindOf(StructuredActivity) then
    self.orderedSubActivities->collectNested(act |
      act.allOrderedSubActivities )->flatten()->asSequence()
  else
    Sequence{self}->union( self.orderedSubActivities )
  endif

context ActivitySequence::initialActivities : Set(Activity)
derive: self.activity->first().initialActivities
```

7.1.8 Definition of Property - allParents : Set

```
context Activity
def: allParents : Set(Activity) =
  self.parent.allParents->flatten()->including(self.parent)
```

7.1.9 Definition of Property - process : BusinessProcess

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

7.1.10 Definition of Property - instantiationActivities : Set

```
context BusinessProcess
def : instantiationActivities : Set(Activity) =
  self.allBasicActivities->select( a |
    a.oclIsKindOf(Receive)
  and
    a.oclAsType(Receive).createInstance )
```

7.1.11 Definition of Property - causalGroups : Set

```
context BusinessProcess
def : instantiationActivities : Set(Activity) =
  self.allBasicActivities->select( a |
    a.oclIsKindOf(Receive)
  and
    a.oclAsType(Receive).createInstance )
```

Chapter 8

Appendix B

8.1 Generics

8.1.1 Introduction

JDK 1.5 introduces several extensions to the Java programming language. One of these is the introduction of generics.

Generics allow us to abstract over types. The most common examples are container types, such as those in the Collection hierarchy.

Here is a typical usage of that sort:

```
List myIntList = new LinkedList(); // 1
myIntList.add(new Integer(0)); // 2
Integer x = (Integer) myIntList.iterator().next(); // 3
```

The cast on line 3 is slightly annoying. Typically, the programmer knows what kind of data has been placed into a particular list. However, the cast is essential. The compiler can only guarantee that an Object will be returned by the iterator. To ensure the assignment to a variable of type Integer is type safe, the cast is required.

What if programmers could actually express their intent, and mark a list as being restricted to contain a particular data type? This is the core idea behind generics. Here is a version of the program fragment given above using generics:

```
List<Integer> myIntList = new LinkedList<Integer>(); // 1
myIntList.add(new Integer(0)); //2
Integer x = myIntList.iterator().next(); // 3
```

Notice the type declaration for the variable myIntList. It specifies that this is not just an arbitrary List, but a List of Integer, written List<Integer>. We say that List is a generic interface that takes a type parameter - in this case, Integer. We also specify a type parameter when creating the list object.

The compiler can now check the type correctness of the program at compile-time. When we say that myIntList is declared with type List<Integer>, this tells us something about the variable myIntList, which holds true wherever and whenever it is used, and the compiler will guarantee it. In contrast, the cast tells us something the programmer thinks is true at a single point in the code. The net effect, especially in large programs, is improved readability and robustness.

Considering an another example from **TAssign.java** generated by JAXB 2.0,

```
protected List<TCopy> _getCopy() {
  if (copy == null) {
    copy = new ArrayList<TCopy>();
  }
return copy;
```

The above function returns a List. With the help of generics, it has been explicitly showed that the copy is an arrayList of type **TCopy** and the function **_getCopy()** returns a list of type **TCopy**. Generics help in defining the return types precisely as its usage is realized, when handled for larger generated code or in larger projects.

8.1.2 WildCards in JAXB 2.0

Let us consider an example of

```
void printCollection(Collection<Object> c)
  { for (Object e : c) {
    System.out.println(e);
  }
}
```

In this example, the **printCollection()** is restricted to get Collection of type **(Object)** alone and cannot accept other types of collections. To overcome that the code can be changed as,

```
void printCollection(Collection<?> c)
  { for (Object e : c) {
    System.out.println(e);
  }
}
```

where **Collection<?>** represents Collection of Unknown.

With the case of JAXB unmarshalling of a bpel document, So what could be the

```
JAXBElement<?> topElem = (JAXBElement<?>) unmarshal
        .unmarshal(new FileInputStream("AmazonFlow.bpel"));
```

supertype of all kinds of JAXBelement? It's written **JAXBElement<?>** (pronounced "JAXBElement of unknown") , that is, a Collection whose element type matches anything. It is called a wildcard type for obvious reasons.

Chapter 9

Appendix C

9.1 Octopus - OCL Tool for Precise Uml Specifications

9.1.1 Introduction

Klasse Objecten has developed Octopus tool to support the use of OCL. It offers two main functionalities

- Octopus can statically check OCL expressions. It checks the syntax, as well as the expression types, and the correct use of model elements like association roles and attributes.
- Octopus is able to transform the UML model, including the OCL expressions, into Java code.

9.1.2 UML/OCL Transformation to Code

Octopus is able to generate a complete 3-tier prototype application from a UML/OCL model.

- **Middle tier** consists of plain old Java objects (POJOs). These POJOs include code for checking invariants and multiplicities from the model. OCL expressions that define the body of an operation are transformed into the body of the corresponding Java method. Derivation rules and initial value specifications are transformed as expected.
- **Storage tier** consists of an XML reader and writer dedicated to the given UML/OCL model. It stores any data content in the prototype application in an XML file. Naturally, it is also able to read the content of this XML file into your prototype application. Furthermore, we may regenerate the application, for instance, because one of the classes was missing an attribute, and the reader will still be able to read the XML file. The reader will read the contents of the XML file and produce objects for whatever classes, attributes, and association ends are still in the model. New model elements simply remain empty.

  The XML file given as input for the UML/OCL model has a predefined XML Schema which remains unconfigurable and reading and writing instances of XML file depends on this Schema file. Approachs to validate any other XML file not compatible with this schema, the manual mapping of data objects must be done.

- **User Interface tier** consists of an implementation of a plug-in for the Eclipse Rich Client Platform. From a Navigator view that shows all instances in the system, it is possible to create and examine instances of the given UML/OCL model. Of course, the invariants or multiplicities of an instance can be checked by pushing a single button.

Bibliography

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

[2] Akehurst D. H., *"Experiment in Model Driven Validation of BPEL Specifications"*, University of Kent at Canterbury, Technical report.

[3] *"Business Process Execution Language for Web Services version 1.1"*, BPEL4WS V1.1 Specification, Technical report.

[4] Keith Mantell, *"From UML to BPEL"*, http://www-128.ibm.com/developerworks/~webservices/library/ws-uml2bpel/, September 2005.

[5] *"The Object Constraint Language(OCL) - Getting Your Models Ready for MDA"*, Second Edition, Addison-Wesley Edition 2003.

[6] *"Introduction to OCTOPUS"*, Klasse Objecten, http://www.klasse.nl/english/overig/index.html.

[7] *"Article: Java Architecture for XML Binding (JAXB 2.0)"*, https://jaxb2-commons.dev.java.net/.

[8] *"Technical Reports on OCL"*, http://www.rspa.com/reflib/FormalMethods.html.

[9] Scott Fordin, *"Article: Java Architecture for XML Binding (JAXB)"*, Sun Developer Network, http://java.sun.com/webservices/jaxb/about.html, October 2004.

[10] Micheal Wahler, *"Using OCL to interrogate your EMF model"*, http://www.zurich.ibm.com/~wah/doc/emf-ocl/index.html, August 2004.

[11] *"Thinking in JAVA"*, Third Edition, Bruce Eckel, A Prentice Hall Edition 2003.

.