Parser and Validator Manual

Nathalie Steinmetz, Holger Lausen

January 31, 2006

Contents

List of Figures

Chapter 1

The Wsml Validator

The Web Service Modeling Language WSML (D16.1v0.21) provides a formal syntax and semantics for the Web Service Modeling Ontology WSMO. WSML consists of a number of variants, namely WSML-Core, WSML-DL, WSML-Flight, WSML-Rule and WSML-Full, based on different logical formalisms (Description Logic, First-Order Logic and Logic Programming).

The Wsml Validator doesn't check for syntax errors, this is done by the WSML parser. Such a syntax error is for example the relation declaration of the following wsml specification:

```
wsmlVariant _"http://www.wsmo.org/wsml/wsml-syntax/wsml-core"

namespace { _"http://www.example.org/ontologies/example#",
  dc   _"http://purl.org/dc/elements/1.1#"}

ontology _"http://www.example.org/ontologies/example"

relation distance
```

Listing 1: WSML Syntax error

Neither cardinality nor parameter types are defined for the relation distance. To correct this error, we define parameter types for this relation:

```
relation distance(ofType City, ofType City, ofType _real)
```

Listing 2: Valid WSML Syntax - invalid WSML-Core

The Wsml Validator checks if this construct is compliant with the different WSML variants. WSML-Core only allows binary relations, so our example is not valid WSML-Core. The Validator throws an ValidationError with an explaining error message.

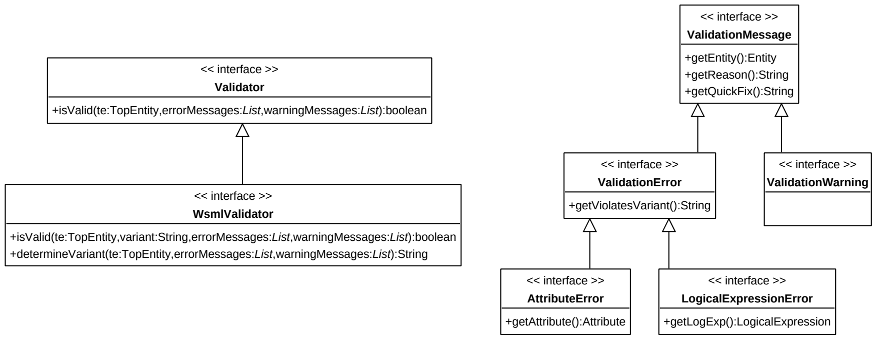
The Wsml Validator checks each Ontology element (concepts, relations, instances, relation instances and axioms) for validity against the WSML variants.

1.1 API

The WsmlValidator extends the Validator interface. It offers two different methods to check whether a TopEntity is valid or not: one indicating the variant against which shall be validated and one without this variant. A third method offers the possibility to determine the variant of a given TopEntity.

A list of errors is filled with the ValidationErrors that occur during the validation. Each error contains information about the Entity in which the error occured, about the reason and about the variant that is violated with this error.

The ValidationError interface presents a hierarchy of errors for a more fine-grained analysis. AttributeErrors and LogicalExpressionErrors can additionally return the violated attribute, respectively logical expression. See figure 1.1.



<< interface >>
Validator
+isValid(te:TopEntity,errorMessages:List,warningMessages:List):boolean
<< interface >>
WsmlValidator
+isValid(te:TopEntity,variant:String,errorMessages:List,warningMessages:List):boolean
+determineVariant(te:TopEntity,errorMessages:List,warningMessages:List):String
<< interface >>
ValidationMessage
+getEntity():Entity
+getReason():String
+getQuickFix():String
<< interface >>
ValidationError
+getViolatesVariant():String
<< interface >>
ValidationWarning
<< interface >>
AttributeError
+getAttribute():Attribute
<< interface >>
LogicalExpressionError
+getLogExp():LogicalExpression

Figure 1.1: Validator Model

1.2 Usage

The following describes the usage of the Wsml Validator.

1.2.1 Creating the Validator

Using the Factory method createWsmlValidator(Map properties) one can create an instance of the WsmlValidator.

The properties map specifies the preferences for the validator initialisation. These preferences can provide a LogicalExpressionFactory to be used within the validator. This factory can be submitted to the properties map as String or as instance of an already existing factory. If it is indicated as String, the WsmlValidator creates the needed factory.

```
WsmlValidator w = Factory.createWsmlValidator(Map properties);
```

Listing 3: Creation of the WsmlValidator

When submitting null as parameter to the create method, a default validator with the default LogicalExpressionFactory is created (default LogicalExpressionFactory: org.deri.wsmo4j.factory.LogicalExpressionFactoryImpl).

1.2.2 Validation process

As can be seen at figure 1.1, the validation can be started via three methods:

Validation methods

The first method - isValid(TopEntity, errors, warnings) - validates against the variant stated in the TopEntity. When no variant is specified in the TopEntity, the TopEntity is checked for valid wsml-full. The second method - isValid(TopEntity, variant, errors, warnings) - validates against the specified variant.

These two methods return either true or false. True means that the TopEntity is valid in the specified variant. False means that it is not valid, and in this case the error reasons can be retrieved from the list of ValidationErrors. In both cases, either when the TopEntity is valid or not, the list of ValidationWarnings can contain warnings.

The third method - determineVariant(TopEntity, errors, warnings) - determines the variant of the given TopEntity. This mustn't necessarly be the variant that is specified in the TopEntity.

The parameters errors and warnings are objects of type `java.util.List` that are filled with ValidationErrors and ValidationWarnings during the validation.

Validity

If a TopEntity is valid in a certain variant, this doesn't mean it is not valid in the other variants. WSML has two alternative layerings, namely WSML-Core → WSML-DL → WSML-Full and WSML-Core → WSML-Flight → WSML-Rule → WSML-Full.

WSML-DL is an extension of WSML-Core and this means that every valid WSML-Core specification is also a valid WSML-DL specification. Every valid WSML-DL specification is also a valid WSML-Full specification. The second layering works analogously.

1.2.3 Validation messages

Two different Validation messages are thrown during the validation: ValidationErrors and ValidationWarnings.

Errors

All the errors that occur during a validation are added to the list of errors. If a TopEntity is not valid in a certain variant, one can find out the reasons by retrieving the ValidationErrors from the list.

The ValidationError interface allows to access different informations about one error. The method getEntity() returns the Entity in which the error occured. The violated variant is retrieved with getViolatedVariant().

The method getReason() returns a textual description of the error. This textual description usually starts with a specified ValidationError code. Those error codes are to be extended.

```
META_MODEL_ERR = "Meta Modelling Error:";
CONC_ERR = "Concept Error";
ATTR_FEAT_ERR = "Attribute Feature Error";
ATTR_CONS_ERR = "Attribute Constraint Error";
ATTR_CARD_ERR = "Attribute Cardinality Error";
REL_ARITY_ERR = "Relation Arity Error";
REL_CONS_ERR = "Relation Constraint Error";
REL_ERR = "Relation Error";
REL_INST_ERR = "Relation Instance Error";
AX_HEAD_ERR = "Axiom - Inadmissible Head formula";
AX_BODY_ERR = "Axiom - Inadmissible Body formula";
AX_FORMULA_ERR = "Axiom - Inadmissible formula";
AX_LHS_ERR = "Axiom - Inadmissible left-hand side formula";
AX_RHS_ERR = "Axiom - Inadmissible right-hand side formula";
AX_ATOMIC_ERR = "Axiom - Inadmissible Atomic formula";
AX_IMPL_BY_ERR = "Axiom - Inadmissible inverse implication formula"
    ;
AX_IMP_ERR = "Axiom - Inadmissible implication formula";
AX_EQUIV_ERR = "Axiom - Inadmissible equivalence formula";
AX_SAFETY_COND = "Axiom - Safety condition doesn't hold for this
    logical expression";
AX_GRAPH_ERR = "Axiom - Invalid Graph";
ID_ERR = "Inadmissible Identifier";
```

Listing 4: ValidationError codes

The method getQuickFix() returns a textual description of a possible quick solution to this error. This method is to be extended.

To allow a more fine-grained analysis, there is a hierarchy of ValidationErrors (see figure 1.1). This is a point to be still extended.

AttributeErrors are thrown by concept attributes. Their method getAttribute() allows to retrieve the actual attribute where the error occured. LogicalExpressionErrors occur in the logical expressions that define an axiom. Their method getLogicalExpression() allows to see in which LogicalExpression the error occured.

Warnings

ValidationWarnings don't affect the validity of a TopEntity. They are meant as warnings and don't necessarily need to be observed. But of course they can represent hints for possible problems.

The methods getEntity(), getReason() and getQuickFix() work the same way as described above for errors.

1.2.4 Usage example

```
public void test() throws Exception {
    createAxiomDef("Man subConceptOf _integer.");
    WsmlValidator w = Factory.createWsmlValidator(null);
    w.isValid(ontology, "http://www.wsmo.org/wsml/wsml-syntax/wsml-
        core",
        errors, warnings);
    ValidationError ve = (ValidationError)errors.firstElement();
    printError(errors);
    assertTrue(ve.getReason().startsWith(ValidationErrorImpl.
        AX_ATOMIC_ERR));
    removeAxiomDef();
}
```

Listing 5: JUnit test - ValidationError retrieving

1.3 Restrictions

The validator is only implemented for ontologies at present.

Chapter 2

Logical Expressions

The WSML syntax consists of two major parts: the conceptual syntax and the logical expression syntax. Logical expressions are used to refine the definitions of WSMO elements using a logical language and occur within axioms.

2.1 API

In the logical expression object model, logical expressions are separated into atomic and compound expressions. Their syntax is inspired by F-Logic.

WSML allows the following logical connectives: **and**, **or**, **implies**, **impliedBy**, **equivalent**, **neg**, **naf**, **forall** and **exists**. As auxiliary symbols are allowed: '(', ')', '[', ']', ',', '=', '**!=**', '**:=:**', **memberOf**, **hasValue**, **subConceptOf**, **ofType** and **impliesType**. WSML allows also the use of the symbols '**:-**' for Logic Programming Rules '**!-**' for database-style constraints.

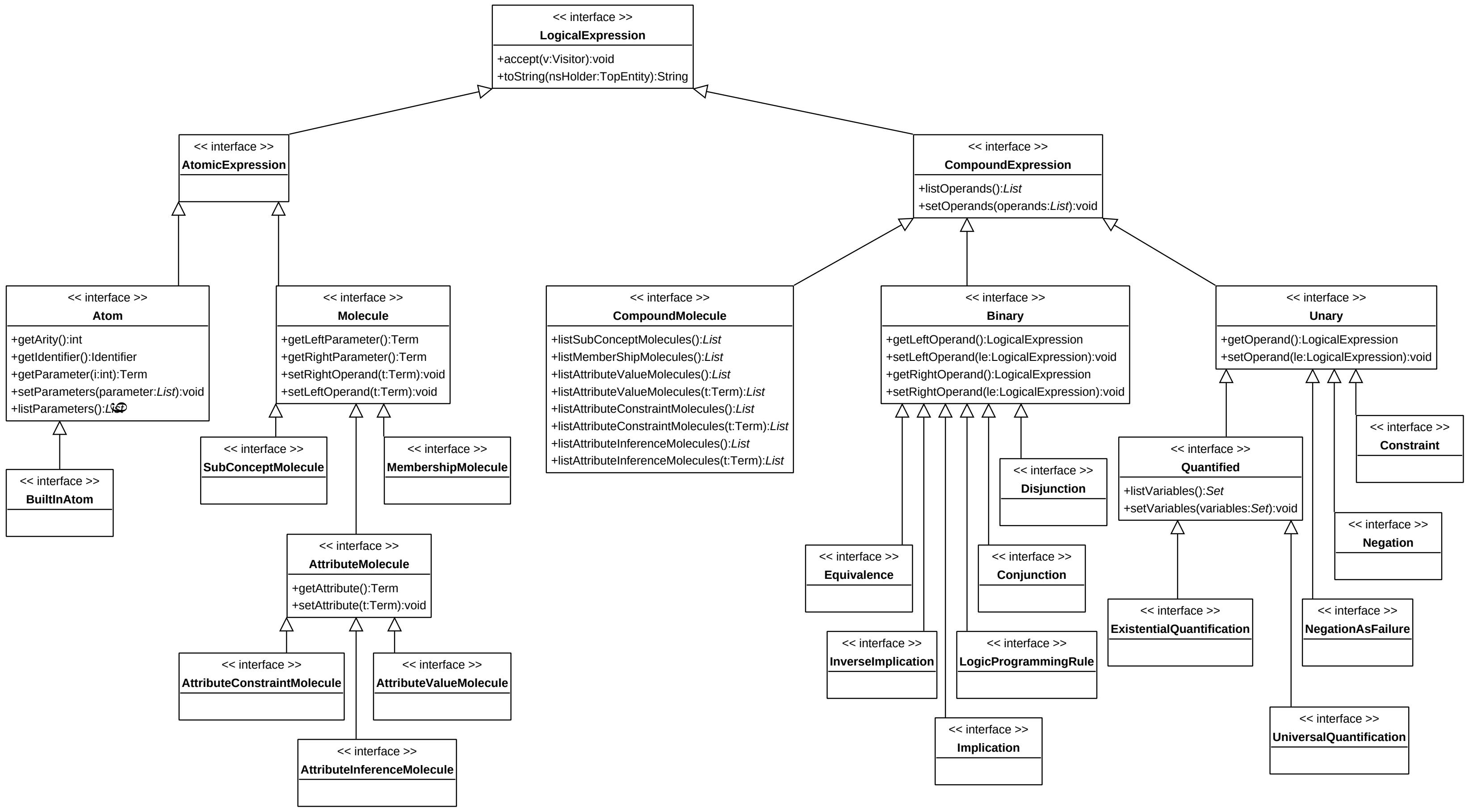
The basic constructs of logical expressions are Terms (see figure 2.2). Based on these terms, atomic and complex formulae are defined (see figure 2.1).

2.2 Parse and serialize Logical Expressions

The Parser is created using the Factory Pattern. A LogicalExpressionFactory, a DataFactory and a WsmoFactory can be passed as parameters. The LogicalExpressionFactory is used to create logical expression objects, the DataFactory to create simple and complex data values, and the WsmoFactory to create wsmo entities.

2.2.1 Parsing in context of TopEntity

When parsing a wsml file, we obtain an array of TopEntities (Ontologies, Goals, Mediators or Web Services). We then cast the first TopEntity to an ontology, which consists of concepts, instances, axioms,... Logical expressions that are



<< interface >>
LogicalExpression
+accept(v:Visitor):void
+toString(nsHolder:TopEntity):String
<< interface >>
AtomicExpression
<< interface >>
CompoundExpression
+listOperands():*List*
+setOperands(operands:*List*):void
<< interface >>
Atom
+getArity():int
+getIdentifier():Identifier
+getParameter(i:int):Term
+setParameters(parameter:*List*):void
+listParameters():*List*
<< interface >>
Molecule
+getLeftParameter():Term
+getRightParameter():Term
+setRightOperand(t:Term):void
+setLeftOperand(t:Term):void
<< interface >>
CompoundMolecule
+listSubConceptMolecules():*List*
+listMemberShipMolecules():*List*
+listAttributeValueMolecules():*List*
+listAttributeValueMolecules(t:Term):*List*
+listAttributeConstraintMolecules():*List*
+listAttributeConstraintMolecules(t:Term):*List*
+listAttributeInferenceMolecules():*List*
+listAttributeInferenceMolecules(t:Term):*List*
<< interface >>
Binary
+getLeftOperand():LogicalExpression
+setLeftOperand(le:LogicalExpression):void
+getRightOperand():LogicalExpression
+setRightOperand(le:LogicalExpression):void
<< interface >>
Unary
+getOperand():LogicalExpression
+setOperand(le:LogicalExpression):void
<< interface >>
BuiltInAtom
<< interface >>
SubConceptMolecule
<< interface >>
MembershipMolecule
<< interface >>
AttributeMolecule
+getAttribute():Term
+setAttribute(t:Term):void
<< interface >>
AttributeConstraintMolecule
<< interface >>
AttributeValueMolecule
<< interface >>
AttributeInferenceMolecule
<< interface >>
Disjunction
<< interface >>
Equivalence
<< interface >>
Conjunction
<< interface >>
InverseImplication
<< interface >>
LogicProgrammingRule
<< interface >>
Implication
<< interface >>
Constraint
<< interface >>
Quantified
+listVariables():*Set*
+setVariables(variables:*Set*):void
<< interface >>
Negation
<< interface >>
ExistentialQuantification
<< interface >>
NegationAsFailure
<< interface >>
UniversalQuantification

Figure 2.1: Logical Expression Model

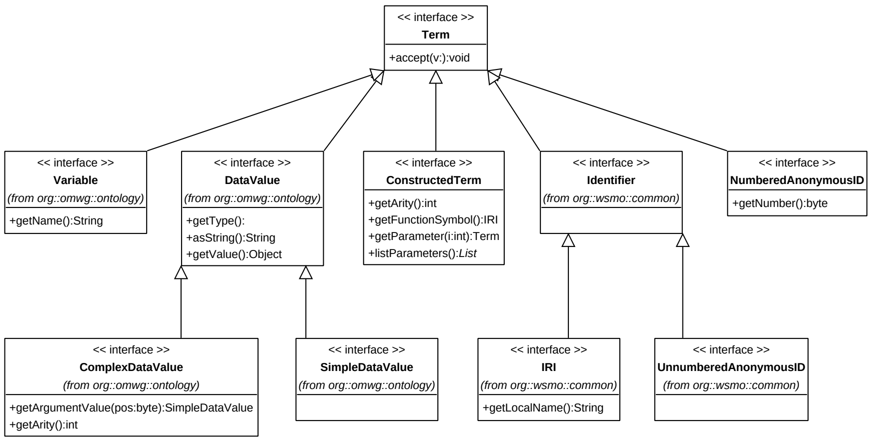


Figure 2.2: Term Model

parsed using the LogicalExpressionFactory, are transformed from a String description to an Object Model.

2.2.2 Create Logical Expressions

To create logical expressions, one needs to use the LogicalExpressionFactory. This one allows to build logical expressions in two different ways. Either from a string, or by building it up directly using different objects like variables, IRIs, molecules, etc. No matter how an expression is created, it will be equal.

Example

```
LogicalExpression  le = leFactory. createLogicalExpression (
      "!− ?x[gender hasValue {?y, ?z}] memberOf Human and ?y = Male and ?z = Female.");
```

Listing 1: Logical expression created from a String.

This axiom means that no human can be both male and female. From this String, the LogicalExpressionFactory builds the object model at figure 2.3.

Create Logical Expressions programmatically

The LogicalExpression Factory interface can also be used to create atomic or compound logical expressions of different types:

```
LogicalExpression createLogicalExpression(String expr);
LogicalExpression createLogicalExpression(String expr, TopEntity
    nsHolder);
Atom createAtom(Identifier id, List params);
CompoundMolecule createCompoundMolecule(List molecules);
```

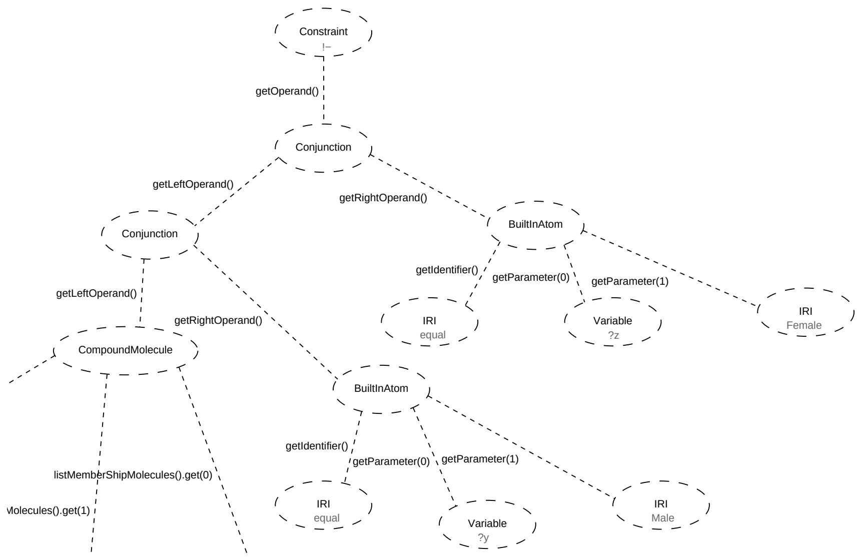


Figure 2.3: Axiom: Part of the Object Model

```
MembershipMolecule createMemberShipMolecule(Term idInstance, Term
     idConcept);
SubConceptMolecule createSubConceptMolecule(Term idConcept, Term
     idSuperConcept);
AttributeValueMolecule createAttributeValue(Term instanceID, Term
     attributeID, Term attributeValue);
AttributeConstraintMolecule createAttributeConstraint(Term
     instanceID, Term attributeID, Term attributeType);
AttributeInferenceMolecule createAttributeInference(Term instanceID
     , Term attributeID, Term attributeType);
Negation createNegation(LogicalExpression expr);
NegationAsFailure createNegationAsFailure(LogicalExpression expr);
Constraint createConstraint(LogicalExpression expr);
Conjunction createConjunction(LogicalExpression exprLeft,
     LogicalExpression exprRight);
Disjunction createDisjunction(LogicalExpression exprLeft,
     LogicalExpression exprRight);
Implication createImplication(LogicalExpression exprLeft,
     LogicalExpression exprRight);
Equivalence createEquivalence(LogicalExpression exprLeft,
     LogicalExpression exprRight);
LogicProgrammingRule createLogicProgrammingRule(LogicalExpression
     exprLeft, LogicalExpression exprRight);
InverseImplication createInverseImplication(LogicalExpression
     exprLeft, LogicalExpression exprRight);
UniversalQuantification createUniversalQuantification(Set variables
     , LogicalExpression expr);
UniversalQuantification createUniversalQuantification(Variable
     variable, LogicalExpression expr);
ExistentialQuantification createExistentialQuantification(Set
     variables, LogicalExpression expr);
ExistentialQuantification createExistentialQuantification(Variable
     variable, LogicalExpression expr);
ConstructedTerm createConstructedTerm(IRI functionSymbol, List
     terms);
```

```
NumberedAnonymousID createAnonymousID(byte number);
```

Listing 2: LogicalExpressionFactory interface.

2.2.3 Modify Logical Expressions

Logical expressions can be modified by modifying the different objects that build a logical expression.

```
LogicalExpression le = leFactory.createLogicalExpression(
      "Susan[ageOfHuman hasValue 31] memberOf Human.", ontology);
CompoundMolecule comp = (CompoundMolecule) le;
AttributeValueMolecule attr = (AttributeValueMolecule) comp.
    listAttributeValueMolecules().get(0);
Identifier id = factory.createIRI("http://ex.org#Mary");
attr.setLeftOperand(id);
```

Listing 3: Modifiying a logical expression.

After the modification of the attribute's left operand, the logical expression looks like follows:

```
Mary[ageOfHuman hasValue 31] memberOf Human.
```

Listing 4: An animal can be both male and female.

2.2.4 Serializing Logical Expressions

A Logical Expression can easily be serialized from an object model to a String again. The overwritten `java.lang.Object` method `toString()` uses the WSML Serializer to serialize the object model to a String.

The method `toString(TopEntity nsHolder)` allows to transmit the namespace container to the Serializer, which is used to abbreviate IRIs to sQNames.

2.3 Visitor Pattern

The visitor design pattern is a way of separating an algorithm from an object structure. This separation allows us to define new operations over existing object structures without modifying those structures, simply by adding a new visitor.

If you have an object class structure, each of these objects has an accept() method that takes a visitor object as an argument. The visitor interface has a different visit() method for each object class. The accept() method of an object class calls the visit() method for its class. Concrete visitor classes can then perform some particular operations over the objects.

2.3.1 Using the visitor pattern

In the wsmo-api there exists a Visitor interface that represents a visitor for the logical expression tree structure and one that represents a visitor for the terms of logical expressions. To use these visitors, you need to write concrete classes that implement the Visitor interfaces.

A concrete visitor class simply fills its visit() methods with the new functionality that this visitor shall offer to the visited objects.

2.3.2 Example

At wsmo4j, the visitor patterns are used, among others, by the Serializer and the WsmlValidator. We are showing the use of the Visitor Patters taking some methods of the WsmlValidator as example:

```
public interface Visitor {
    void visitAtom(Atom expr);
    void visitCompoundMolecule(CompoundMolecule expr);
    ...
}
```

Listing 5: Visitor interface with two visit() methods.

```
public class WsmlCoreExpressionValidator implements Visitor {
    ...
    public void visitAtom(Atom expr) {
        if (expr.getArity() != 2) {
            addError(expr, ValidationError.AX_ATOMIC_ERR +
                ":\nThe atom must be a binary atom\n" +
                leSerializer.serialize(expr));
        }
    }
    public void visitCompoundMolecule(CompoundMolecule expr) {
        Iterator i = expr.listOperands().iterator();
        while(i.hasNext()){
            ((Molecule)i.next()).accept(this);
        }
    }
    ...
}
```

Listing 6: Concrete visitor class implementing Visitor interface.