Meta-Model Evaluation and Initialization

A software metrics tool that follows the framework of SPMDL would need to compute or define ways to compute the corresponding meta-model elements. For DMM, the meta-model evaluator comes in the following format, implementing several "built-in" meta-model queries:

```
class DMMMetaModelEvaluater : MetaModelEvaluater
{
    /* retrieve that implement relations of the DMM model */

    StructuralElement[]     Get_Accesses (BehaviouralElement b) { … }
    SourcePart[]    Get_Contains (SourceObject so) { … }
    ModelObject[]   Get_Declares (SourceObject so) { … }
    ModelObject[]   Get_Defines  (SourceObject so) { … }
    Comment[]       Get_Describes (SourceObject so) { … }
    Value[]  Get_HasValue (Variable v) { … }
    Package[]       Get_Imports (Class c) { … }
    SourceFile[]    Get_Includes (SourceFile sf { … })
    Class[]  Get_InheritsFrom (Class c) { … }
    BehaviouralElement[]    Get_Invokes (BehaviouralElement be) { … }
    …

}
```

Table 2 lists all required meta-model queries for the DMM model.

Table 2 – Built-in SPMDL Queries based in the DMM Model

Return Type	Query	Description
StructuralElement[]	Get_Accesses (BehaviouralElement b)	Returns structural elements that the given behavioral element accesses.
SourcePart[]	Get_Contains (SourceObject so)	Returns SourcePart elements "Contained" in the given SourceObject.
ModelObject[]	Get_Defines (SourceObject so)	Returns ModelObject's that are defined in the given SourceObject.
Comment[]	Get_Describes (SourceObject so)	Returns comments associated with the given SourceObject.
Value[]	Get_HasValue (Variable v)	Returns a list of values of the given Variable.
Package[]	Get_Imports (Class c)	Returns Package's imported by a given class.
SourceFile[]	Get_Includes (SourceFile sf)	Returns file's included by a given source file.
Class[]	Get_InheritsFrom (Class c)	Returns super-classes of a given class.
BehaviouralElement[]	Get_Invokes (BehaviouralElement be)	Returns a list of behavioral elements (e.g. methods) invoked by a given element.
Invokes[]	Get_ActualParameterOf (ModelElement me)	Returns the actual parameters list of a given element.
Type[]	Get_DefinedInTermsOf (Type t)	Returns the type used in the definition of a given type, e.g. coupling through ADT.
EnumeratedType[]	Get_EnumerationLiteralOf (EnumeratedLiteral el)	Returns literals if a given enumeration literals list.
Field[]	Get_FieldsOf (StructuredType st)	Returns fields of a given structure.
Method[]	Get_MethodsOf (Class c)	Returns the list of methods declared within a class.
Type	Get_TypeOf (Value v)	Returns the Type of a given Value object.
FormalParameter[]	Get_ParameterOf (BehaviouralElement)	Returns the list of parameters defined in a given BehavioralElement.
Type	Get_ReturnTypeOf (BehavioralElement be)	Returns the Type of a given BehavioralElement.
Package[]	Get_SubpackagesOf (Package p)	Returns sub-packages of a given Package.

In order to satisfy the performance goals, through progressive computation, the Visitor design pattern has been used. Metric definitions represented in SPMDL correspond to objects that perform the actual evaluation for the metrics. In order to provide a full incremental implementation, the software parser takes each artifact and passes its information to metrics evaluators where they get called every time a software element is ready for evaluation. The following class, the AbstractVisitor, is a base-class for all the metrics evaluators. It consists of metrics visiting methods that are called when the corresponding program element is parsed. The class also contains helper methods for declaring and updating values in the meta-model intermediate database. Metric evaluators implement the portions necessary to compute values of the metrics following the visitor's pattern event model.

```
class AbstractVisitor {
     public abstract void visitPackage(string packageName);
     public abstract void visitClass (string className);
     public abstract void visitMethod (string methodName);
     public abstract void visitField (string fieldName);
     public abstract void visitValue (string valueName);
     public abstract void visitVariable      (string variableName);
     public abstract void visitType  (string TypeName);
     public abstract void visitEnumerationType             (string EnumerationTypeName);
     public abstract void visitStructuredType(string StructuredTypeName);
     public abstract void visitFormalParameter             (string FormalParameterName);
     public abstract void visitRoutine       (string RoutineName);
     public abstract void visitExecutableValue             (string ExecutableValueName);
     public abstract void visitCollectionType(string CollectionTypeName);

     // fullElementName refers the the full qualified name of the object
     // (e.g. package.class.method.variable)
     public void declareVariable(string varType, string varScope, string
          fullElementName)
     {
          // register the variable in the temp store and
          // associate it with the given scope
          Store.createVarvarType, varScope, fullElementName);
     }

     public string retrieveVariable(string varScope, string elementName)
     {
          return Store.getVarValue(varScope, fullElementName);
     }

     public string updateVariable(string varScope, string elementName, string
          newValue)
     {
          return Store.setVarValue(varScope, elementName, newValue);
     }

     public boolean evaluateCondition(string condition, string operator, string
          expectedValue)
     {
          return Store.evaluate(condition, operator, expectedValue);
     }
}
```

All element names represent full qualified names of the static elements. For example the method C in class B of package A should be referred to as "A.B.C".

The parser algorithm therefore is as follows:

```
class Parser
{
     private AbstactVisitor visitor;
     void parse(Class c)   {
          visitor.visitClass(c.name);
          /* visit methods of the given class */
          for (Method m : c.methods)
          {
                /* visit method parameters */
                for (FormalParameter p : m.parameters)
                      visitor.visitFormalParameter(p.name);
                /* visit the actual method */
                visitor.visitMethod(m.name);
```

```
                /* visit variables used in the method */
                for (Variable v : m.variables)
                      visitor.visitVariable(v.name);
                /* visit other classes accesed in this method */
                for (Type t : m.accesses)
                      visitor.visitType(t.name);
          }
          /* visit fields of the given class */
          for (Field f : c.fields)
          {
                visitor.visitField(f.name);
                visitor.visitValue(f.value.name);
          }
          for (Method m : c.methods)
                visitor.vistiMethod(m.name);
    }
}
```

Therefore, the major role of SPMDL under this incremental evaluation is to describe the implementation algorithm of each visiting method in order to come up with the final metric value.

As an example, consider the following expression which can be used for computing the number of public methods in a given class. The metric query can be written in SPMDL as (dmmQuery refers to a meta-model query that implements the DMM model):

```
<dmmQuery>
    <description>Compute the number of methods in a given class
    </description>
    <visitor scope="class">
          <variable name="numMethods" type="long" scope="class" />
    </visitor>
    <visitor scope="method">
          <condition expression="isPublic = false" action="continue" />
          <math:expression>
                <linkLong name="numMethods"/>
                <add datatype="long">
                            <long value="1"/>
                      </add>
                </linkLong>
          </math:expression>
    </visitor>
</dmmQuery>
```

This expression would declare a variable called "methodCount" in the scope of the current class. The declared variable is therefore used in an XMLMath expression to update the value of the variable after each visit.

This SPMDL representation is essentially equivalent to the following code (which would be generated during the actual parsing of the metric definition). Notice that the Adapter design pattern is applied here through the VisitorAdapter class in order to avoid implementing all methods of the AbstractVisitor:

```
class ConcreteVisitor : VisitorAdapter
{
    public abstract void visitClass         (string className)
    {
         declareVariable("long", "class", className+".numMethods");
    }
    public abstract void visitMethod (string methodName)
    {
         if (evaluateCondition("isPublic", "equals", "true"))
         {
              long temp = Long.parse(retrieveVariable("class",
                    methodName+".numMethods"));
              updateVariable("class", methodName+".numMethods", temp + 1);

         }
    }
```

```
}
```

Invalidation criteria can also be described in SPMDL as in the following example:

```
<dmmQuery>
    <visitor scope="class">
         <variable name="numMethods" type="long" scope="class" />
    </visitor>
    <visitor scope="method">
         <condition expression="isPublic = false" action="continue" />
         <invalidationCriteria affectedElement="Method" condition="isPublic =
               True" scope=”class” />
         <math:expression>
               <linkLong name="numMethods"/>
                     <add datatype="long">
                         <long value="1"/>
                   </add>
               </linkLong>
         </math:expression>
    </visitor>
</dmmQuery>
```

The variable "numMethods" is declared under the scope of the current class in the temporary store. When methods of the given class are being evaluated, the current value is retrieved and incremented before writing back to the store. This example invalidates all elements of type “Method” which satisfy the condition “isPublic = true” within the scope of the “class”.