

# Formalizing Object-Oriented Design Metrics upon the UML Meta-Model

**Aline Lúcia Baroni**  
École des Mines de Nantes, France  
FCT/Universidade Nova de Lisboa  
Departamento de Informática  
2825-114 Monte da Caparica, Portugal  
[aline@di.fct.unl.pt](mailto:aline@di.fct.unl.pt)

**Fernando Brito e Abreu**  
Information Systems Group (INESC)  
FCT/Universidade Nova de Lisboa  
Departamento de Informática  
2825-114 Monte da Caparica, Portugal  
[fa@di.fct.unl.pt](mailto:fa@di.fct.unl.pt)

## Abstract

*This paper discusses the formalization effort of object-oriented design metrics definitions and presents some concrete examples, developed upon the UML meta-model. The Object Constraint Language (OCL), a part of the Unified Modeling Language (UML) standard, is used in the formalization. The combination of the UML meta-model and OCL allows unambiguous metrics definition, which in turn helps increasing tool support for object-oriented metrics extraction. This formalization renders possible the comparisons among different sets of metrics, as well as it may be used to establish a common vocabulary among different stakeholders. As consequence, the precision of the metrics collection increases, contributing to the overall quality of the Software Engineering process.*

## 1. Introduction

The lack of formalization has been felt for a long time in the object-oriented modeling area [1, 2]. For instance, in the first well-known book [3] on the subject of metrics for the object-oriented paradigm most proposed metrics were defined in natural language. As an improvement, some authors have used a combination of set theory and simple algebra to express their metrics [4-7], but the mathematical background may not be easy to grasp.

Consider, for example, the measure of distance  $\delta_M$ , defined by Poels [8] as the average distance between the object types of two different non-empty dynamic conceptual schemes. The notion behind this metric may be defined and interpreted in many ways, according to distinct viewpoints. What is a distance? Which are the conditions for measurement? Is the distance expressed by some degree of dissimilarity?

To avoid the ambiguity generated by the informal definition, Poels presents the mathematical development of the measure as follows:

$$\delta_M(M_P, M_Q) = 0 \Leftrightarrow M_P \Delta M_Q = \phi$$

$$\delta_M(M_P, M_Q) = \frac{\sum_{i=1}^I \sum_{j=1}^J \delta(P_i, Q_j)}{I.J} \Leftrightarrow M_P \Delta M_Q \neq \phi$$

$$\delta_M(M_P, M_Q) = \left[ \frac{\sum_{i=1}^I \sum_{j=1}^J \delta_{alph}(P_i, Q_j)}{I.J}, \frac{\sum_{i=1}^I \sum_{j=1}^J \delta_{atr}(P_i, Q_j)}{I.J} \right] \Leftrightarrow M_P \Delta M_Q \neq \phi$$

$$\left[ \frac{\sum_{i=1}^{I'} \sum_{j=1}^{J'} \delta_{seq}(P_i, Q_j)}{I.J}, \frac{\sum_{i=1}^I \sum_{j=1}^J \delta_{data}(P_i, Q_j)}{I.J} \right]$$

where:

$M_P$  and  $M_Q$  are non-empty dynamic conceptual schemes;

$M_P \Delta M_Q = \emptyset \Leftrightarrow (\forall P \in M_P, \exists Q \in M_Q : \delta(P, Q) = 0) \wedge (\forall Q \in M_Q, \exists P \in M_P : \delta(Q, P) = 0)$ ;

cardinality( $M_P$ ) =  $I$ ;                      cardinality( $M_Q$ ) =  $J$ ;

$P_i \in M_P$              $i = 1, \dots, I$ ;     $P_j \in M_Q$              $j = 1, \dots, J$ .

The above definition can be used for both scalar and vector representations of the measure  $\delta(P_i, Q_j)$  – second and third definitions respectively. However, inferring the meaning of this formula, even knowing each of the components involved, is not an easy nut to crack.

The previous examples show that two problems can arise from the formality degree used to define metrics, namely the informal (or natural language) definition problem and mathematical formal definition problem [9]. The former can generate diverge results, as people using metrics can interpret them in several ways. The latter requires a strong mathematical background to cope with the expressions complexity, which most of software practitioners may not have.

In this work, we propose an approach for defining design metrics that combines understandability and formality. Our approach is verified and validated for sake of correction and for guaranteeing the quality of the formalizations. The remaining part of this article is organized as follows. In section 2, the Object Constraint Language is briefly introduced. In

section 3, the formalization of metrics is explained and exemplified for the MOOD2 metrics set [10]. Section 4 shows the architecture that supports the ideas being proposed. Finally, section 5 outlines our conclusions and further work.

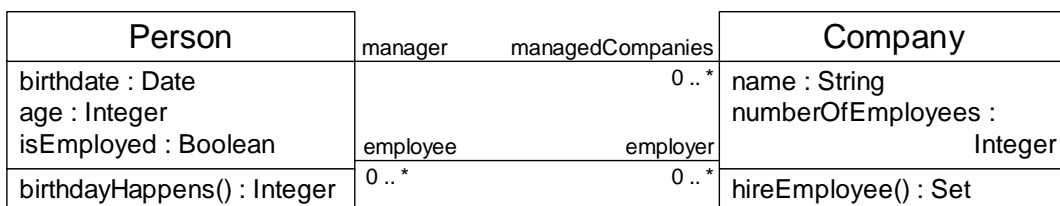
## 2. Formalization using the Object Constraint Language (OCL)

Several attempts have been made to combine accuracy and non-ambiguity with object-oriented modeling. The most recent and promising approach, which bridges formal methods with object-orientation is called OCL - Object Constraint Language, and it is a part of the UML standard, published by the OMG (Object Management Group) [11].

OCL is a formal, yet simple notation, to be used jointly with UML diagrams and whose syntax has some similarities to those of object-oriented languages such as Smalltalk, C++ or Eiffel. It is underpinned by mathematical set theory and logic, as it happens with formal languages, but it was designed for usability and to be easily grasped by anybody familiar with object-oriented modeling and UML notation in particular. OCL allows expressing three kinds of constraints on parts of object-oriented models: *invariants*, *pre-conditions* and *post-conditions*.

Invariants are constraints representing conditions that must be met by all instances of the class, during their existence. Their context is, therefore, a class. Pre-conditions are constraints that must be true for an operation to be executed and they traduce the obligations to be fulfilled by the object requiring the service. Post-conditions are constraints that must be true when the operation ends its execution and they traduce the obligations to be fulfilled by the object that offers the service. The context of both pre and post-conditions is an operation, as in the followings extracts from the simple model of figure 1:

<u><i>Person::birthdayHappens()</i></u> <i>post: age = age@pre + 1</i>	<u><i>Person</i></u> <i>self.age &gt; 18</i>	<u><i>Company::hireEmployee(p : Person)</i></u> <i>pre: not employee -&gt; includes (p)</i>
---	---	--



**Figure 1 – A simple example**

In the context\* of the *Person*, *self* refers to an instance of *Person*. In the expression *self.age*, *self* is an instance of the type *Person*. The invariant  $self.age > 18$  states that a person to be employed cannot be under aged (the minimum age in this case is 18).

In the post-condition example, the property *age* refers to the feature of *Person* after completion of operation. The property *age@pre* refers to the value of the feature *age* of the *Person* before the execution of the operation *birthdayHappens*.

The pre-condition on the operation *hireEmployee* exhibits one example of navigation through a diagram association. It means that a *Person* can only be employed by the *Company* if he/she is not already employed by the *Company*.

OCL constraints are free of side effects. This means that the state of the objects does not change by the application of an OCL expression. Expressions can range from simple comparisons (e.g. an attribute having an upper limit) to complex navigations in a class diagram through their associations.

Constraints convey a number of benefits, including precision and design documentation, resulting in better (unambiguous) communication among the involved parts, such as designers, users, programmers, testers and managers and, since OCL is a typed language, it is possible to check constraints for validity during modeling.

### 3. Formalization of Object-Oriented Metrics

In order to achieve precision into design, designers have to understand the semantics of the modeling concepts, usually held by a *meta-model*. A meta-model is a language for describing models, which explains the semantics of model objects and their relationships. For instance, the UML meta-model [12] is the description of UML elements, produced with the UML notation itself [13].

In this work we use the UML meta-model as a basis for the extraction of quantitative data (metrics), which are formalized with OCL. Even when expressed mathematically, some metrics may be ill defined.

The ill definition problem may happen due to two reasons:

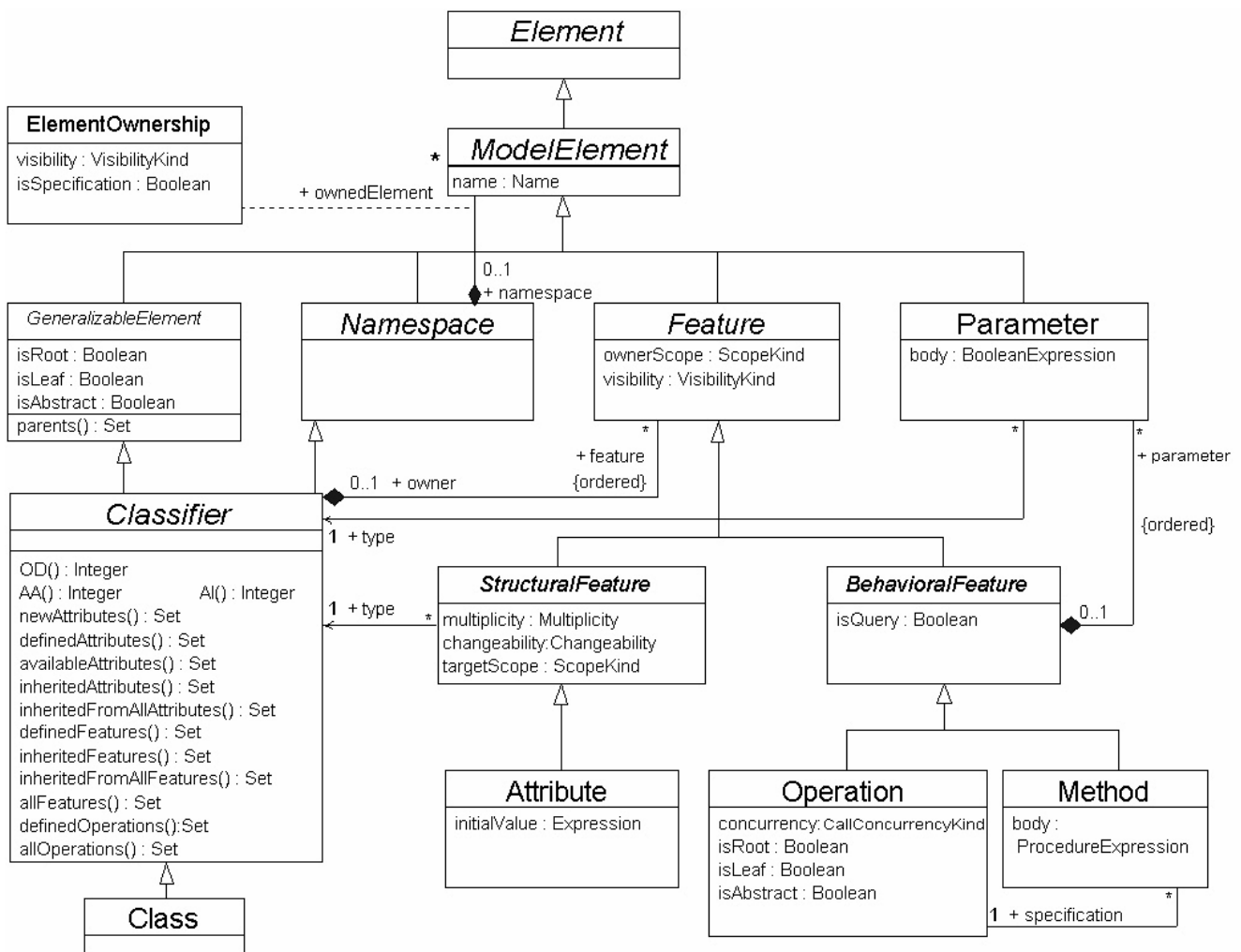
- i) Metrics definitions are usually presented without the corresponding context, that is, without expressing which is the corresponding meta-model where the entities of interest and their interrelationships are expressed;

---

\* In this article, the context of the expression is written in underlined letters.

- ii) Metrics definition is done without an underlying formal specification approach that uses the former meta-model as the contextual input. This formal specification should specify under which conditions the metrics are applicable.

We formalized the set of the MOOD2 metrics [10, 14] using the GOODLY [14, 15] and the UML meta-model as background. Now, we want to formalize other sets using the UML meta-model. As an example, we present the formalization of two metrics.

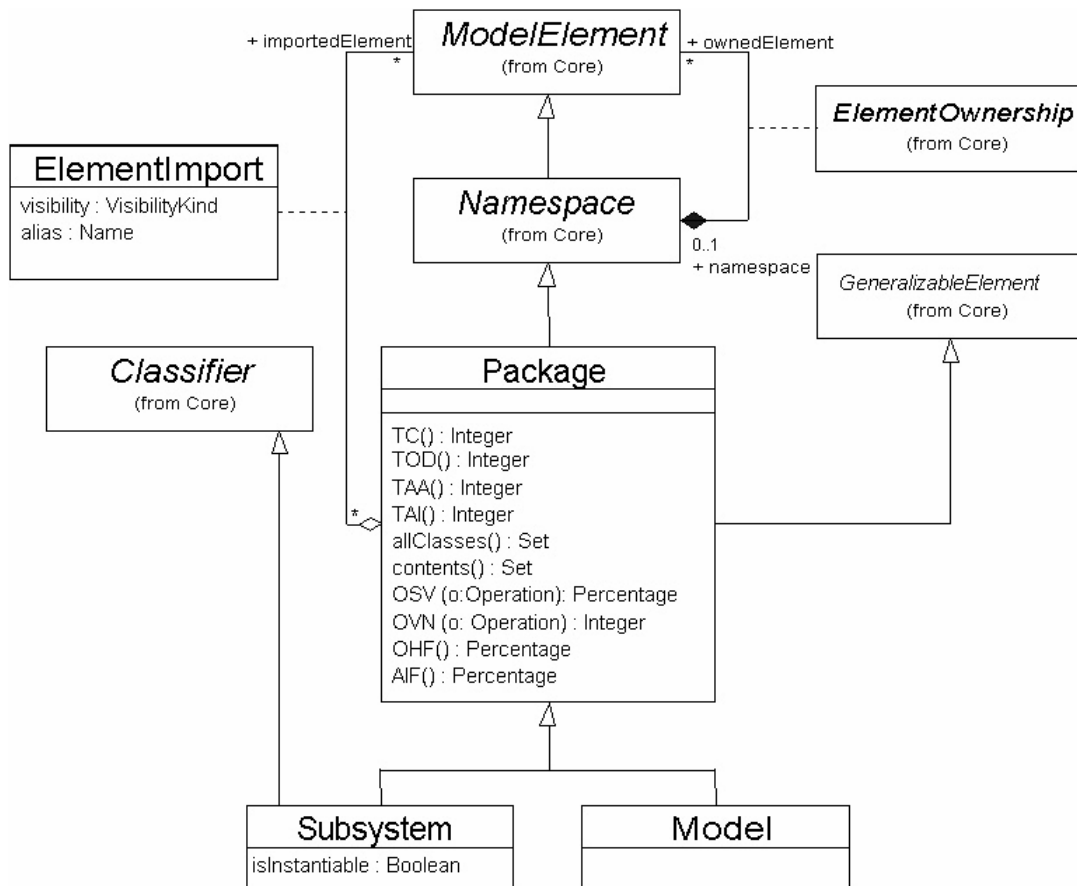


**Figure 2 – Part of the UML Meta-Model Core Package**

The parts of the UML meta-model illustrated in figures 2 and 3 are used to produce and validate around 100 auxiliary functions that are used to express the metrics. The limitations on the metric applicability are defined with OCL pre-conditions, while the metric result is defined as an operation or as an OCL post-condition. Two examples are

demonstrated below, for the metrics OHF (Operations Hiding Factor) and AIF (Attribute Inheritance Factor) [10].

To calculate the metrics, several auxiliary functions are created. These functions are part of a general library that can be used in the formalization of other metrics.



**Figure 3 – Part of the UML Meta-Model Model Management Package**

OHF was originally defined as “*the quotient between the sum of the invisibilities of all operations defined in all classes in the current specification and the total number of operations defined in the specification*”. The invisibility of one operation is the percentage from the total number of classes belonging to the specification where the operation in context is not visible. On one hand, when one operation is *public*, its invisibility is null. If all the operations are public, the numerator goes to zero, that is  $OHF() = 0$ . On the other hand, when one operation is *private*, its invisibility is unitary. If all the operations are private the numerator turns equal to the denominator and  $OHF() = 1$ .

To calculate OHF, the auxiliary functions *TC*, *TOD*, *OD*, *definedOperations*, *definedFeatures*, *allClasses*, *contents*, *allOperations*, *allFeatures*, *parents*, *OSV* and *OVN* are necessary. They are explained and formalized below.

<i>Name</i>	<b><i>TC – Total Classes</i></b>
<i>Informal Definition</i>	<i>Total number of Classes in the Package.</i>
<i>Formal Definition</i>	<b><i>Package:: TC(): Integer</i></b> <i>post: result = allClasses() -&gt; size()</i>

<i>Name</i>	<b><i>TOD – Total Operations Defined</i></b>
<i>Informal Definition</i>	<i>Total number of defined Operations in the Package.</i>
<i>Formal Definition</i>	<b><i>Package:: TOD(): Integer</i></b> <i>post: result = allClasses() -&gt; iterate ( elem: Class; acc: Integer = 0   acc + elem.OD() )</i>

<i>Name</i>	<b><i>OD – Operations Defined</i></b>
<i>Informal Definition</i>	<i>Number of Operations defined in the Classifier.</i>
<i>Formal Definition</i>	<b><i>Classifier:: OD(): Integer</i></b> <i>post: result = definedOperations() -&gt; size()</i>

<i>Name</i>	<b><i>definedOperations</i></b>
<i>Informal Definition</i>	<i>Set of Operations that belong to the current Classifier. Excludes inheritance.</i>
<i>Formal Definition</i>	<b><i>Classifier:: definedOperations(): Set( Operation )</i></b> <i>post: result = self.definedFeatures() -&gt; select ( f   f.ocIsKindOf ( Operation ) ) -&gt; collect ( f   f.ocAsType ( Operation ) ) -&gt; asSet</i>

<i>Name</i>	<b><i>definedFeatures</i></b>
<i>Informal Definition</i>	<i>Set of Features defined in the Classifier. Excludes inheritance.</i>
<i>Formal Definition</i>	<b><i>Classifier:: definedFeatures(): Set( Operation )</i></b> <i>post: result = self.feature -&gt; asSet</i>

<i>Name</i>	<b><i>allClasses</i></b>
<i>Informal Definition</i>	<i>Set of all Classes belonging to the current Package.</i>
<i>Formal Definition</i>	<b><i>Package:: allClasses(): Set( Class )</i></b> <i>post: result = self.contents() -&gt; iterate ( elem: ModelElement; acc: Set ( Class ) = ocEmpty ( Set ( Class ) )   elem.ocIsTypeOf ( Class ) implies acc -&gt; union ( acc -&gt; including ( elem.ocAsType ( Class ) ) ) )</i>

Name	<b>contents</b>
Informal Definition	Set containing the ModelElements owned by or imported by the Package.
Formal Definition	<b><u>Package:: contents(): Set( ModelElement )</u></b> post: result = self.ownedElement -> union ( self.importedElement )

Name	<b>allOperations</b>
Informal Definition	Set containing all Operations of the Classifier itself and all its inherited Operations.
Formal Definition	<b><u>Classifier:: allOperations(): Set( Operation )</u></b> post: result = self.allFeatures() -> select( f   f.ocIsKindOf ( Operation ) ) -> collect ( f   f.ocAsType ( Operation ) ) -> asSet

Name	<b>allFeatures</b>
Informal Definition	Set containing all Features of the Classifier itself and all its inherited Features.
Formal Definition	<b><u>Classifier:: allFeatures(): Set( Feature )</u></b> post: result = self.feature -> asSet -> union ( self.parents() -> collect ( g   g.ocAsType ( Classifier ).allFeatures() ) -> flatten -> asSet )

Name	<b>parents</b>
Informal Definition	Set containing all direct parents of the Element.
Formal Definition	<b><u>GeneralizableElement:: parents(): Set( GeneralizableElement )</u></b> post: result = self.specialization -> collect ( c   c.child ) -> asSet() -> excluding ( self )

Name	<b>OSV – Operation to Specification Visibility</b>
Informal Definition	Percentage of Classes in the considered Package where the Operation can be accessed (excludes the class where the operation is declared).
Formal Definition	<b><u>Package:: OSV( o: Operation ): Percentage</u></b> pre: self.TC() > 1 post: result = ( self.OVN ( o ) - 1 ) / ( self.TC() - 1 )
Comments	The pre-condition states that at least one more Class must exist, besides the one where the Operation is declared.

Name	<b>OVN – Operation Visibility Number</b>
Informal Definition	Number of Classes in the considered Package where the Operation can be accessed.
Formal Definition	<b><u>Package:: OVN( o: Operation ): Integer</u></b> post: result = self.allClasses() -> iterate ( elem: Class; acc: Integer = 0   if elem.OCV ( o ) then acc + 1 else acc endif )

<i>Name</i>	<b><i>OCV – Operation to Classifier Visibility</i></b>
<i>Informal Definition</i>	<i>Indicates if a Classifier can access the Operation.</i>
<i>Formal Definition</i>	<b><u>Classifier:: OCV( o: Operation ): Boolean</u></b> <i>post: result = self.allOperations() -&gt; exists (</i> <i>( o.owner = self ) or</i> <i>( o.visibility = #public ) or</i> <i>( ( o.visibility = #protected ) and</i> <i>( self.ascendants().oclAsType( Classifier ).allOperations()</i> <i>-&gt; includes( o ) ) )</i> <i>or ( ( self.allOperations() = ocEmpty( Set ( Operation ) ) )</i> <i>and ( o.visibility = public ) )</i>
<i>Comments</i>	<i>One Classifier can access the Operation “o” when:</i> <ul style="list-style-type: none"> <li>- <i>It is the owner of the Operation;</i></li> <li>- <i>The Operation is public;</i></li> <li>- <i>The Operation is protected and belongs to one ascendant of the current Classifier;</i></li> <li>- <i>It has no Operation. If the “o” is public, all Classes in the Package can access it, even empty Classes (that, in practice, do not exist).</i></li> </ul>

Finally, using these functions we can define the metric OHF as follows:

<i>Name</i>	<b><i>OHF – Operations Hiding Factor</i></b>
<i>Informal Definition</i>	<i>Quotient between the sum of the invisibilities of all Operations defined in all Classes in the current Package and the total number of Operations defined in the Package.</i>
<i>Formal Definition</i>	<b><u>Package:: OHF(): Percentage</u></b> <i>pre: self.TC() &gt; 1</i> <i>pre: self.TOD() &gt; 0</i> <i>post: result = allClasses().allOperations() -&gt; asSet()</i> <i>-&gt; iterate ( elem: Operation; acc: Real=0  </i> <i>acc + 1 - self.OSV( elem ) ) / self.TOD()</i>

AIF was originally defined as the “quotient between the number of inherited attributes in all classes of the specification and the number of available attributes for all classes of the current specification”. The set of available attributes of one class is the union of the locally defined ones with the inherited attributes, but not redefined or cancelled. When there is no effective inheritance (there are no hierarchies) or when all the attributes are redefined, the numerator becomes null and  $AIF() = 0$ . When the specification has only classes without their own attributes, but with some inherited ones, then the numerator becomes equal to the denominator and  $AIF() = 1$ .

To calculate AIF, the auxiliary functions *TAA*, *AA*, *availableAttributes*, *newAttributes*, *definedAttributes*, *inheritedFromAllAttributes*, *inheritedAttributes*, *inheritedFromAllFeatures*, *inheritedFeatures*, *TAI* and *AI* are necessary. The metric also requires the functions *allClasses*, *definedFeatures* and *parents* above explained.

<i>Name</i>	<b><i>TAA – Total Attributes Available</i></b>
<i>Informal Definition</i>	<i>Total number of available Attributes in the Package.</i>
<i>Formal Definition</i>	<b><i>Package:: TAA(): Integer</i></b> <i>post: result = allClasses() -&gt; iterate( elem: Class; acc: Integer = 0 /  acc + elem.AA() )</i>

<i>Name</i>	<b><i>AA – Attributes Available</i></b>
<i>Informal Definition</i>	<i>Number of Attributes that may be associated to instances of the Classifier.</i>
<i>Formal Definition</i>	<b><i>Classifier:: AA(): Integer</i></b> <i>post: result = availableAttributes() -&gt; size()</i>

<i>Name</i>	<b><i>availableAttributes</i></b>
<i>Informal Definition</i>	<i>Attributes that may be applied to instances of the Classifier.</i>
<i>Formal Definition</i>	<b><i>Classifier:: availableAttributes(): Set( Attribute )</i></b> <i>post: result = newAttributes() -&gt; union ( inheritedFromAllAttributes() )</i>

<i>Name</i>	<b><i>newAttributes</i></b>
<i>Informal Definition</i>	<i>Attributes defined in the Classifier that are not overriding inherited ones.</i>
<i>Formal Definition</i>	<b><i>Classifier:: newAttributes(): Set( Attribute )</i></b> <i>post: result = definedAttributes() – inheritedFromAllAttributes()</i>

<i>Name</i>	<b><i>definedAttributes</i></b>
<i>Informal Definition</i>	<i>Set of Attributes defined in the Classifier.</i>
<i>Formal Definition</i>	<b><i>Classifier:: definedAttributes(): Set( Attribute )</i></b> <i>post: result = self.definedFeatures() -&gt; select ( f / f.ocIsKindOf ( Attribute ) ) -&gt;  collect ( f / f.ocAsType ( Attribute ) ) -&gt; asSet</i>

<i>Name</i>	<b><i>inheritedFromAllAttributes</i></b>
<i>Informal Definition</i>	<i>Attributes that the current Classifier inherits, until the top of the hierarchy.</i>
<i>Formal Definition</i>	<b><i>Classifier:: inheritedFromAllAttributes(): Set( Attribute )</i></b> <i>post: result = self.inheritedAttributes()  -&gt; union ( self.inheritedFromAllFeatures() )  -&gt; select ( f / f.ocIsKindOf ( Attribute ) )  -&gt; collect ( f / f.ocAsType ( Attribute ) ) -&gt; asSet</i>

<i>Name</i>	<b><i>inheritedAttributes</i></b>
<i>Informal Definition</i>	<i>Number of direct inherited Attributes.</i>
<i>Formal Definition</i>	<b><i>Classifier:: inheritedAttributes(): Set( Attribute )</i></b> <i>post: result = self.inheritedFeatures() -&gt; select ( f / f.ocIsKindOf ( Attribute )</i> <i>-&gt; collect ( f / f.ocAsType ( Attribute ) ) -&gt; asSet</i>

<i>Name</i>	<b><i>inheritedFromAllFeatures</i></b>
<i>Informal Definition</i>	<i>Features that the current Classifier inherits, until the top of the hierarchy.</i>
<i>Formal Definition</i>	<b><i>Classifier:: inheritedFromAllFeatures(): Set( Attribute )</i></b> <i>post: result = self.inheritedFeatures() -&gt; union ( self.parents() -&gt;</i> <i>collect ( p / p.ocAsType( Classifier ).inheritedFromAllFeatures() )</i> <i>-&gt; flatten -&gt; asSet)</i>

<i>Name</i>	<b><i>inheritedFeatures</i></b>
<i>Informal Definition</i>	<i>Direct inherited Features from the Classifier.</i>
<i>Formal Definition</i>	<b><i>Classifier:: inheritedFeatures(): Set( Attribute )</i></b> <i>post: result = self.parents() -&gt; iterate( elem: GeneralizableElement;</i> <i>acc: Set( Feature ) = ocEmpty( Set( Feature ) )  </i> <i>acc -&gt; union ( elem.ocAsType( Classifier ).definedFeatures() ) )</i>

<i>Name</i>	<b><i>TAI – Total Attributes Inherited</i></b>
<i>Informal Definition</i>	<i>Total number of Attributes inherited in the Package.</i>
<i>Formal Definition</i>	<b><i>Package:: TAI(): Integer</i></b> <i>post: result = allClasses() -&gt; iterate ( elem: Class; acc: Integer = 0  </i> <i>acc + elem.AI() )</i>

<i>Name</i>	<b><i>AI – Attributes Inherited</i></b>
<i>Informal Definition</i>	<i>Number of all inherited attributes.</i>
<i>Formal Definition</i>	<b><i>Classifier:: AI(): Integer</i></b> <i>post: result = inheritedFromAllAttributes() -&gt; size()</i>

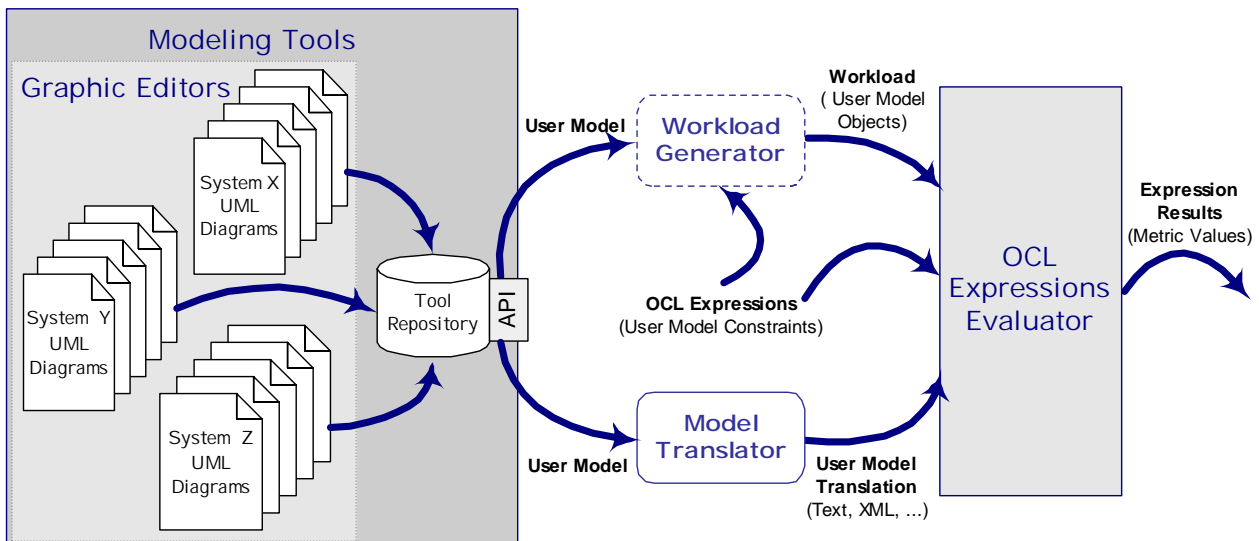
AIF is then, formalized as:

<i>Name</i>	<b><i>AIF – Attributes Inheritance Factor</i></b>
<i>Informal Definition</i>	<i>Quotient between the number of inherited attributes in all classes of the package and the number of available attributes (locally defined plus inherited) for all classes of the current package.</i>
<i>Formal Definition</i>	<b><i>Package:: AIF(): Percentage</i></b> <i>pre: self.TAA() &gt; 0</i> <i>post: result = self.TAI() / self.TAA()</i>

In a similar way, all the 16 metrics of MOOD and MOOD2 are formalized. Our next steps consist of formalizing and testing other sets, to finally compare them.

#### 4. An Architecture for Metrics Formalization

The current state of the art for adding precision to object-oriented modeling by the use of OCL is depicted in figure 4. Commercial UML modeling tools (as Rational Rose [16], Objectory [17], JDeveloper [18], QuickUML [19], PowerDesigner [20], etc.) provide some graphic diagram editors that allow building models of systems. The models – represented by X, Y and Z in the picture – are stored in the tool repository.



**Figure 4 – Model Level Architecture**

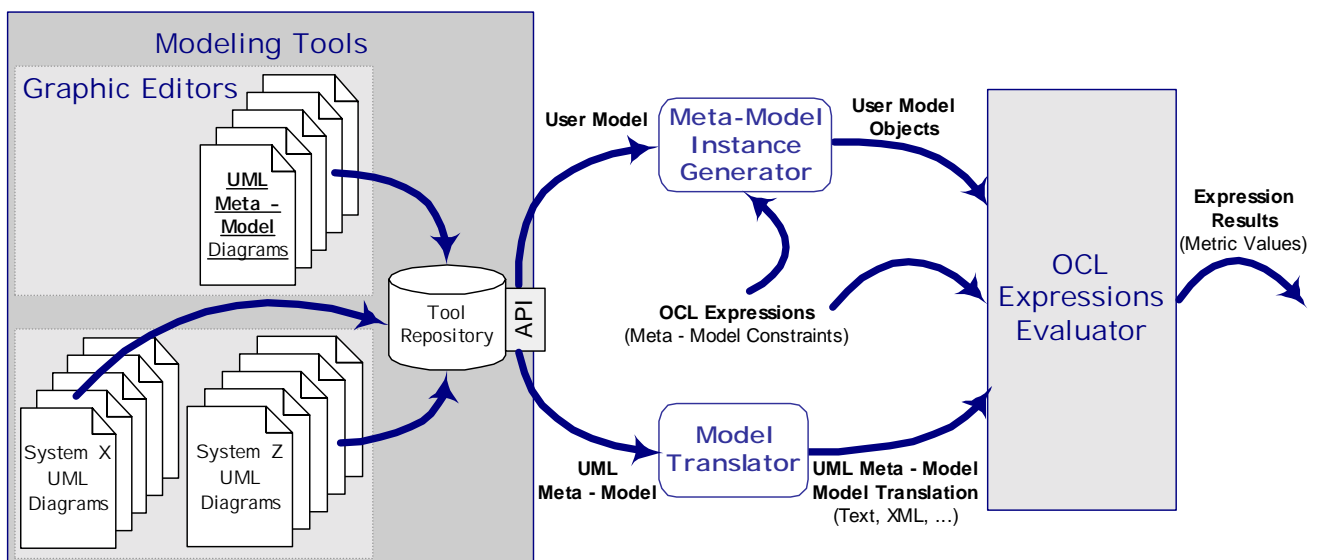
Nowadays, modeling tools do not offer facilities for the evaluation of OCL expressions over the models in the repository. Nevertheless, several tools – like Use [21], Cybernetic Parser [22], Elixer [23], ModelRun [24], etc – are emerging from undergoing research projects and can be used to formalize the models, provided that they can be exported with an appropriate input format to the OCL tools. Typically, a textual file representing the model is generated by a translator (XML can be used as an example to represent the model).

After the file conversion of the model (to a representation that can be understood by OCL tools), the real instances of the entities in the diagram are created and the model is populated (i.e., a plenty of objects, corresponding to the entities in the model, are created). These instances are the base of the assertions that are constructed with OCL. For this process

one workload generator tool (see figure 4) would be of great help because, frequently, the UML model instances are done “by hand”.

The diagrams that compose the models and their respective objects serve as input to an OCL evaluation tool, which takes the converted representation of the diagram, the added OCL constraints and the instances of the model, and evaluate each of the constraints, showing the results. The OCL evaluator should be capable of verifying if the constraints are broken or not, for a given workload of user model instances. Moreover, it should evaluate each assertion separately and to provide feedback on which are the design test cases that meet or break the constraints.

While the architecture depicted in figure 4 corresponds to a model level evaluation, the one depicted in figure 5 is related to a meta-model level evaluation.



**Figure 5 – Meta-Model Level Architecture**

In the meta-model level architecture, all the functionalities of the model-level are preserved. Notwithstanding, there are two main additions: one is the introduction of the class diagrams corresponding to the UML meta-model. Another is the introduction of an automatic instance generator, which will take the meta-model and automatically generate all the instances to populate it. Using these features (meta-model and corresponding instances) we formalize and test several design sets of metrics that can be found in the literature, expressed as OCL expressions upon the UML meta-model, as shown in the previous section.

A textual version (in XML format for example) of the UML meta-model can be obtained from a UML meta-model class diagram, using the architecture represented in figure

5. We developed a meta-model instance generator to instantiate the objects corresponding to the meta-classes. Currently, we have formalized and tested the validity of the MOOD and MOOD2 metrics [25, 26].

We plan to formalize other well-known design metrics sets. This effort will help to clarify the metrics definitions and to assess their suitability to measure UML designs. Furthermore, the later effort will enable us to propose a quality model for metrics, which will consequently facilitate the creation of the meta-metrics – metrics that measure metrics characteristics. Some examples of those characteristics, still illustrated in an informal way, are:

- Understandability: the effort required to understand the metric. It is inversely proportional to the weighted sum of the meta-model classes and associations involved in the metric definition.
- Efficiency: number of resources necessary to compute the metric. It is inversely proportional to the computational complexity of the metric calculation algorithm (as expressed in OCL).

Our work helps improving quality in the models, since it helps developers (at an initial stage of the software production) to estimate important characteristics of the system while the quality factors are being evaluated and adapted for getting a better product.

## 5. Conclusions and further work

We used the OCL, a part of the UML standard, to define object-oriented design metrics in a very natural and understandable way. The precision granted by the formality of OCL comes at a much lower cost, for both practitioners and tool builders, than when using other formal specification constructs. Since UML became a *de facto* standard, both in academia and industry, more and more people are expected to use OCL in their designs and, as such, to understand its syntax and semantics.

Besides formalizing some metrics sets using the UML meta-model, we also plan to make a similar effort based upon the OML (*OPEN Modeling Language*) meta-model. OML emerged from the OPEN (*Object-oriented Process, Environment and Notation*) consortium [32-35]. The latter is supported by a large group of well-known methodologists such as *Brian*

*Henderson-Sellers* (author of the *MOSES* method [31, 32]), *Ian Graham* (author of *SOMA – Semantic Object Modelling Approach* [33]), *Donald Firesmith* [34] and *Jim Odell* [35].

Moreover, we plan to abstract the common characteristics of those sets in order to build a high level meta-model. The latter will be a framework for describing, classifying and accessing existing metric sets, as well as a basis for the production of new ones. Our idea is to introduce a framework that allows practitioners to build new metric sets or improve old ones and afterwards, to create the meta-metrics-model. We also expect to detect the limitations of the metrics during the formalization process.

We believe that our efforts can indeed contribute to Software Engineering practical aspects, providing better tool support for metrics and also to emphasize the importance of quantitative approaches on industry and academic applications.

## 6. References

- [1] B. Meyer, “On Formalism in Specifications”, *IEEE Software*, vol. 2, pp. 6-26, 1985.
- [2] Y. Wand, “A Proposal for a Formal Model of Objects”, in *Object-Oriented Concepts, Databases, and Applications*, K. Won and F. H. Lochovsky, Eds. Reading, MA, EUA: Addison-Wesley, 1989.
- [3] M. Lorenz and J. Kidd, *Object-Oriented Software Metrics: A Practical Guide*. Englewood Cliffs, NJ, EUA: Prentice Hall, 1994.
- [4] S. R. Chidamber and C. F. Kemerer, “A Metrics Suite for Object Oriented Design”, *IEEE Transactions on Software Engineering*, vol. 20, pp. 476-493, 1994.
- [5] F. B. Abreu and R. Carapuça, “Object-Oriented Software Engineering: Measuring and Controlling the Development Process”, presented at 4th International Conference on Software Quality, McLean, Virginia, EUA, 1994.
- [6] F. B. Abreu, “Quantitative Methods for Object-Oriented Systems”, presented at 7th ERCIM Workshop on Object Oriented Databases, Lisboa, Portugal, 1995.
- [7] B. Henderson-Sellers, “The Mathematical Validity of Software Metrics”, *Software Engineering Notes*, pp. 89-94, 1996.
- [8] G. Poels and G. Dedene, “Formal Software Measurement for Object-Oriented Business Models”, presented at 7th European Software Control and Metrics Conference (ESCOM'96), Wilmslow, UK, 1996.
- [9] A. L. Baroni, S. Braz, and F. B. Abreu, “Using OCL to Formalize Object-Oriented Design Metrics Definitions”, in *ECOOP'02 Workshop on Quantitative Approaches in OO Software Engineering, Lecture Notes in Computer Science*: Springer-Verlag, 2002.
- [10] F. B. Abreu, “The MOOD2 Metrics Set (in Portuguese)”, INESC, April 1998.
- [11] OMG, Rational Software et al, “Object Constraint Language Specification (version 1.1)”, Menlo Park, CA, EUA: Object Management Group, 1997.
- [12] OMG, “Unified Modeling Language Specification (version 1.3)”, Rational Software et al., Ed.: Object Management Group, 1999.
- [13] OMG, “UML Notation Guide (version 1.4)” , R. e. al., Ed. Menlo Park, CA, EUA: Object

- Management Group, 2001.
- [14] F. B. Abreu, "Using OCL to formalize object oriented metrics definitions," INESC, ES007/2001, May 2001.
  - [15] F. B. Abreu, L. M. Ochoa, and M. A. Goulão, "The GOODLY Design Language for MOOD Metrics Collection", INESC, R16/97, March 1997.
  - [16] Rational Software Corporation, "Rational Rose", <http://www.rational.com/products/rose/index.jsp>, 2001.
  - [17] Rational Software Corporation, "Rational Objectory Case Tool", <http://www.inf.ufsc.br/poo/ine5383/orydemo/ory.htm>, 4.1 ed, 1998.
  - [18] Oracle Technology Network, "Oracle 9i JDeveloper", <http://otn.oracle.com/products/jdev/content.html>, 9i ed, April 2002.
  - [19] Excel Software, "QuickUML", <http://www.excelsoftware.com/index.html>
  - [20] Sybase Inc., "PowerDesigner", <http://www.sybase.com/products/enterprisemodeling/powerdesigner>, 9.0 ed.
  - [21] University of Bremen, "USE - A UML-based Specification Environment", <http://dustbin.informatik.uni-bremen.de/projects/USE/>
  - [22] Cybernetic Intelligence GmbH, "OCL Compiler", <http://www.cybernetic.org/>, 1.5 ed.
  - [23] Elixir Technology, "Elixir Java IDE", <http://www.elixirtech.com/>, 2.4 ed.
  - [24] BoldSoft, "ModelRun", <http://www.boldsoft.com/products/modelrun/index.html>, BoldSoft MDE AB, Sweden.
  - [25] F. B. Abreu, "Metrics for Object Oriented Software Development", presented at 3rd International Conference on Software Quality, Lake Tahoe, Nevada, EUA, 1993.
  - [26] F. B. Abreu and J. S. Cucho, "Collecting and Analyzing the MOOD2 Metrics", presented at ECOOP'98 Workshop - Object-Oriented Product Metrics for Software Quality Assessment, Brussels, Belgium, 1998.
  - [27] D. Firesmith, B. Henderson-Sellers, and I. Graham, "OPEN Modeling Language (OML) - Core Notation Specification", OPEN Consortium Version 0.1, July 1996.
  - [28] B. Henderson-Sellers and I. Graham, "OPEN: Towards Method Convergence?", *IEEE Computer*, vol. 29, pp. 86-89, 1996.
  - [29] B. Henderson-Sellers, "The OPEN Methodology", *Object Magazine*, vol. 6, pp. 56-59, 1996.
  - [30] B. Henderson-Sellers, T. Simons, and H. Younessi, *The OPEN Toolbox of Techniques*: Addison-Wesley Publishing Company, 1998.
  - [31] B. Henderson-Sellers, *A BOOK of Object-Oriented Knowledge*. Sydney, Australia: Prentice Hall PTR, 1991.
  - [32] B. Henderson-Sellers and J. M. Edwards, *BOOK TWO of Object-Oriented Knowledge: the Working Object*. Sydney, Australia: Prentice Hall, 1994.
  - [33] I. M. Graham, "A Non-Procedural Process Model for Object-Oriented Software Development", *Report on Object Oriented Analyses and Design*, vol. 5, 1995.
  - [34] D. G. Firesmith and B. Henderson-Sellers, *The OPEN Process Framework: an Introduction*: Addison-Wesley Publishing Company, 2000.
  - [35] J. Odell, "Meta-Modeling", presented at OOPSLA'95 Metamodeling Workshop, 1995.