Assessing UML Model Quality by Utilizing Metrics

Khanh-Hoang Doan
Computer Science Department
University of Bremen
28359 Bremen, Germany
doankh@informatik.uni-bremen.de

Martin Gogolla
Computer Science Department
University of Bremen
28359 Bremen, Germany
gogolla@informatik.uni-bremen.de

***Abstract*—Utilizing metrics is an efficient mechanism to measure and evaluate the quality of models, since many well-known and accepted software metrics have been successfully transferred from the code level to the model level. In a modeling context, OCL seems to be a natural and qualified choice for expressing them, because of its expressiveness and its availability in modeling tools. The metrics employed in this contribution will be defined on a metamodel level, and their evaluation will be done in an automatically generated metamodel instantiation that reflects the model under consideration. The employed metrics cover both the class scope and the model scope, and software designers can define their own metrics. We introduce a complete process for model quality assessment with pre-defined metrics. In addition, a metrics configuration defined by an experienced chief designer can be translated into OCL invariants and be evaluated to give quality feedback to software designers relieving them from detailed OCL expertise. Our approach has been successfully integrated into a software design tool.**

***Index Terms*—Metrics, UML Metamodel, OCL, Model quality assessment**

I. INTRODUCTION

In software engineering, it is widely accepted that the quality of a software system should be guaranteed in the early phases of the development process. The problems occurring in the early stages, e.g., the design phase, of software system development often yield many faults in later stages, e.g., the coding phase, where it is costly to identify and fix them. Recently, Model-Driven Engineering (MDE) has become more and more popular since it promises an increase in the efficiency and quality of software development. In early phases of development, modeling languages such as the UML (Unified Modeling Language) [1] enriched by the OCL (Object Constraint Language) [2] play an increasingly important role and become the key elements in the software development process. Therefore, model quality has significant influence on the outcome of software. To measure and evaluate the quality of models, utilizing metrics can be an efficient approach since many well-known and accepted software metrics have been successfully transferred from the code level to the model level. Actually, multiple authors have proposed many sets of software metrics in the literature [3]–[8], and a considerable number of them are applicable to the model level. Basically, these metrics can be used to measure internal design quality characteristics, e.g., complexity or coupling. A survey of the theoretical validation as well as empirical validation of UML class model metrics in [9] has proved the applicability of metrics in practice.

The discussion in [10] has shown that OCL seems to be a potential candidate for expressing metrics of UML models because of its support for all mathematical operations needed for metrics formulation and its availability in modeling tools. In the literature, there are several works that present approaches using OCL at the metamodel level for model metrics definition [10], [23]–[25]. These works introduced approaches for metrics formulated in OCL, but they did not focus how to utilize metrics defined with OCL for evaluating designs with the help of constraints as well as for uncovering smells in models. Additionally, the discussion in [10] indicated that one of the drawbacks of OCL usage is the low degree of developer acceptance. In order to make the usage of metrics defined by OCL more acceptable for developers, who might be not familiar with OCL, we present a three level metamodel framework and a method for metrics definition and utilize a pre-defined metrics library for UML model assessment. To achieve that, we add the *full OMG (Object Management Group) UML 2.4 metamodel* to the topmost level of a traditional two level modeling approach. To offer access to the meta level, a metamodel instantiation which reflects the user model, i.e., the model to be measured, is automatically generated and added to the middle level, i.e., on the same level as the user model. On this basis, the metrics will be defined on the metamodel level and their evaluation is done on the metamodel instantiation level. The details of our metrics definition approach within the metamodel, for both class scope and model scope metrics, are presented here as well. Following this approach and the given examples, software designers can define their own metrics.

Assessing quality properties of a UML class model, e.g., design properties, respecting naming conventions or other properties, is important for fault detection in the early stage of the software development process. We introduce a complete process for model quality assessment with pre-defined metrics. A metrics evaluation configuration given by an experienced chief designer will be translated into OCL invariants and evaluated to give qualified feedback to software designers. Further inspection of the elements violating the quality requirements is included in the assessment process. In technical terms, our approach has been successfully integrated into the tool USE (UML-based Specification Environment) [11], [12],

an originally two level modeling tool, which supports OCL parsing, testing, validation and verification.

We classify the research contribution of our works as follows.

- We propose a metamodeling framework that makes the user model in form of a metamodel instantiation with direct access explicitly available to the user.
- We offer an approach for interactive metrics definition in which existing metrics can be modified until a desired new metric has been developed.
- The approach offers a transparent metric-based method for model assessment and design smell detection without the need for designers to have OCL expertise.
- The paper presents a successful tooling of metrics definition and metric-based model assessment for UML and OCL models.

The rest of the contribution is structured as follows. In Sect. II, we present a discussion of internal software quality and a list of selected metrics that are essential to measure the internal quality. Section III introduces our metamodeling approach and how it can be exploited for metrics measurement. The proposal for model quality assessment by checking metrics with corresponding thresholds in constraints is presented in Sect. IV. The contribution ends with concluding remarks and future work in Sect. VI.

II. UML Model Quality Measurement with Metrics

A. Internal Software Quality

The meaning of the notion "software quality" is different based on the views of different user groups. For end users, a high quality system must be simple to use, run fast and be reliable. Meanwhile, developers expect the system should be easy to maintain, reuse and adapt to requirement changes, for example.

In principle, software quality is evaluated through observation of different external attributes. **External quality attributes** are those that can be measured only with respect to how the software product relates to its environment. For example, a system is considered to have *poor reliability*, if it does not perform as expected. According to [13], software quality attributes can be stated as follows: functionality, reliability, usability, efficiency, maintainability and portability. It is clear that in order to measure external attributes, information about the software itself is not enough. Thus, external attributes can only be measured late in the software development process. However, it is costly to identify the problems and to fix them during the external quality assessment at later phases. Therefore, approximate evaluation of external qualities by assessing internal qualities could be a potential alternative.

Internal quality attributes are those that can be measured by considering the product itself. Utilizing the information about the product itself, for example, the design model, one can measure several structural properties such as size or coupling. These internal measurements, are not directly meaningful to the product qualities. However, because of the causal impact of the internal qualities on the external qualities, the measurement of internal qualities could provide a vision of the overall outcome quality of the product. Actually, a number of empirical studies indicated in the survey [9] have proved that optimizing certain internal qualities (e.g., reducing the complexity of the design) can lead to several particular desirable external qualities (e.g., increasing the maintainability of the product). Extracting internal qualities of the class model is a kind of early measurement.

B. Class Model Metrics

For a long time, metrics have been proposed in order to determine class model quality. Table I shows a collection of selected metrics known from the literature. These metrics have already been defined within our work, and are ready to be used for model quality assessment in our tool. The number next to the name indicates the reference source, in which the metric was first defined. We classify the metrics by (a) the quality aspects they measure, i.e., complexity, coupling, object-oriented design principles, size, and (b) the context where they are applicable, i.e., class scope or model scope. Please note that cohesion metrics are not included in the list because measuring them is mostly impossible at the model level since implementation details are needed.

Table I
SELECTED UML CLASS MODEL METRICS.

Metric [source]	Type	Scope
Total number of classes in the design (DSC) [14]	Size	Model
Number of total associations (NAssoc) [8]	Size	Model
Number of attributes per class (NOA) [15]	Size	Class
Number of local methods (NOM) [16]	Size	Class
SIZE2 = NOA + NOM [16]	Size	Class
Average Parameters per Method (APPM) [4]	Size	Class
Number of children (NOC) [3]	Complexity	Class
Number of hierarchies (NOH) [14]	Complexity	Model
Depth of inheritance (DIT) [3]	Complexity	Class
Maximum depth of inheritance (MaxDIT) [3]	Complexity	Model
Specialization index (SIX) [4]	Complexity	Class
Number of methods inherited (NMI) [4]	OO design	Class
Number of methods overridden (NMO) [4]	OO design	Class
Attribute inheritance factor (AIF) [5]	OO design	Model
Method inheritance factor (MIF) [5]	OO design	Model
Polymorphism factor (PF) [5]	OO design	Model
Abstractness (A) [17]	OO design	Model
Coupling between object (CBO) [3]	Coupling	Class
Number of attributes that have another class as their type (DAC) [16]	Coupling	Class
Number of different classes that are used as types of class attributes (DAC') [16]	Coupling	Class
Number of associations (NAS) [7]	Coupling	Class
Direct class coupling (DCC) [14]	Coupling	Class

In next section, we will introduce how these metrics can be measured with a metamodel approach and how our approach is integrated into a two level modeling tool.

III. Measuring Metrics using Metamodeling

A. Metamodeling

The idea of our three level modeling approach as shown in Fig. 1 has been presented in [18]. In this section we have to shortly recapitulate this approach, (a) which gives the background framework for our proposed method of metric definition and model assessment and (b) which is needed in order to precisely introduce the new notions and the new features in later sections (e.g., the option to interactively develop new metrics or to assess models with thresholds for developers without extensive OCL expertise). To make our three level approach more clear and self-contained, we define major terms in our approach as follows:

- *User model*: a UML and OCL class model that describes the structure of a system to be modeled. The major elements of a UML and OCL class model are: classes, attributes, operations (or methods), associations, and invariants.
- *Metamodel*: the UML metamodel (the OMG superstructure) [20] that provides the notions to define UML class models. This *metamodel* itself is an explicit UML and OCL class model.
- *Metamodel instantiation*: an automatically generated instantiation of the metamodel that reflects and is equivalent to the *user model*.
- *Metaclass*: a class of the metamodel.
- *Metaobject*: an instance of a metaclass.

The OMG has defined the Meta-Object-Facility (MOF) [19] as a fundamental standard for modeling. MOF provides a four level architecture for system modeling (three of them are shown in Fig. 1).

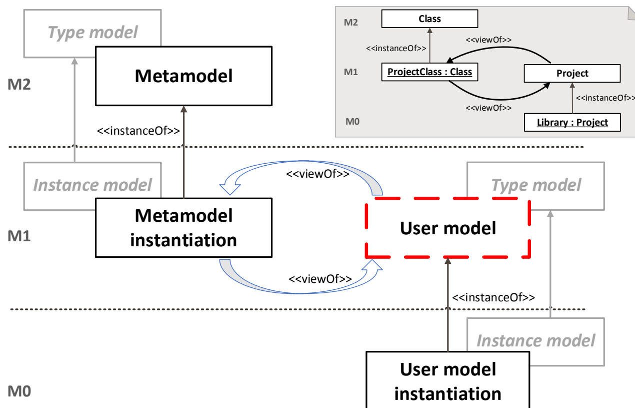

Figure 1. General schema for three level modeling.

Based on the MOF architecture, we now make the third OMG level M2 explicitly available. Figure 1 shows the general schema for our three level modeling approach. The

Table II
RELATIONSHIP BETWEEN USER MODEL ELEMENTS AND METAMODEL ELEMENTS.

User model element	Related meta-model classe	Related UML metamodel association
Class	Class	
Attribute	Property	$Class_{class} - Property_{ownedAttribute}$ $Property_{typedElement} - Type_{type}$
Association	Association	$Class_{endType} - Association_{association}$
Association End	Property	$Association_{association} - Property_{memberEnd}$ $Association_{association} - Prop.ty_{navigableOwnedEnd}$ $Association_{association} - Property_{ownedEnd}$
Operation	Operation	$Class_{class} - Operation_{ownedOperation}$ $Operation_{typedElement} - Type_{type}$
Parameter	Parameter	$Operation_{operation} - Parameter_{ownedParameter}$ $Parameter_{typedElement} - Type_{type}$
Generalization	Generalization	$Class_{subClass} - Class_{superClass}$ $Generalization_{Generalization} - Class_{specific}$ $Generalization_{Generalization} - Class_{general}$
Redefined Attribute/ Redefined Association End		$Property_{property} - Property_{redefinedProperty}$
Subsetted Attribute/ Subsetted Association End		$Property_{property} - Property_{subsettedProperty}$

model at level M2 is the *metamodel*. This metamodel itself is an explicit UML class model formulated in USE with (currently) 63 classes and 99 associations. It is pre-loaded as a metamodel for all user models at level M1 and is fixed during the modeling process. In the middle of our three level modeling approach there is a *user model* at level M1. This is the central artifact in our metamodeling approach, highlighted by the dashed rectangle in Fig. 1. The key point that makes our approach ready for defining class model metrics and model evaluation is a *metamodel instantiation* added to the M1 level. This *metamodel instantiation* contains a number of *metaobjects*. Each *metaobject* is an instance of a *metaclass* and the character of the generated *metaobjects* and links is determined by the *user model*. For example, if there are two classes in the *user model*, then two *metaobjects* instantiated from *metaclass* `Class` are generated. Each *metaobject* is named as a combination of the name of the corresponding element from the *user model* and the corresponding *metaclass* from which it is instantiated. A simple example of our three level modeling approach is presented in the right upper corner of Fig. 1. The metaobject `ProjectClass` is an instance of *metaclass* `Class` and its name is the combination of 'Project', the name of the class from *user model*, and 'Class', the name of the *metaclass*. Our approach visits all user model elements (e.g., classes, attributes, operations, associations) and generates the corresponding *metaobjects* and links. Table II shows the mapping between the *user model* elements and the related *metaclasses* and associations, which are the type elements for the generated *metaobjects* and links. In this work, we utilize the USE specific language SOIL (Simple OCL-based Imperative Language) [21] to create these *metaobjects* and links.

To provide a better understanding of our approach on defining metrics on the metamodel, the central part of the UML metamodel which presents important elements for our work is shown in Fig. 2. All metrics are defined by OCL expressions based on the class diagram in that figure. As a forward reference, the OCL expression in Fig. 4 and Fig. 5 rely on this class diagram. In addition, this *metamodel* is pre-

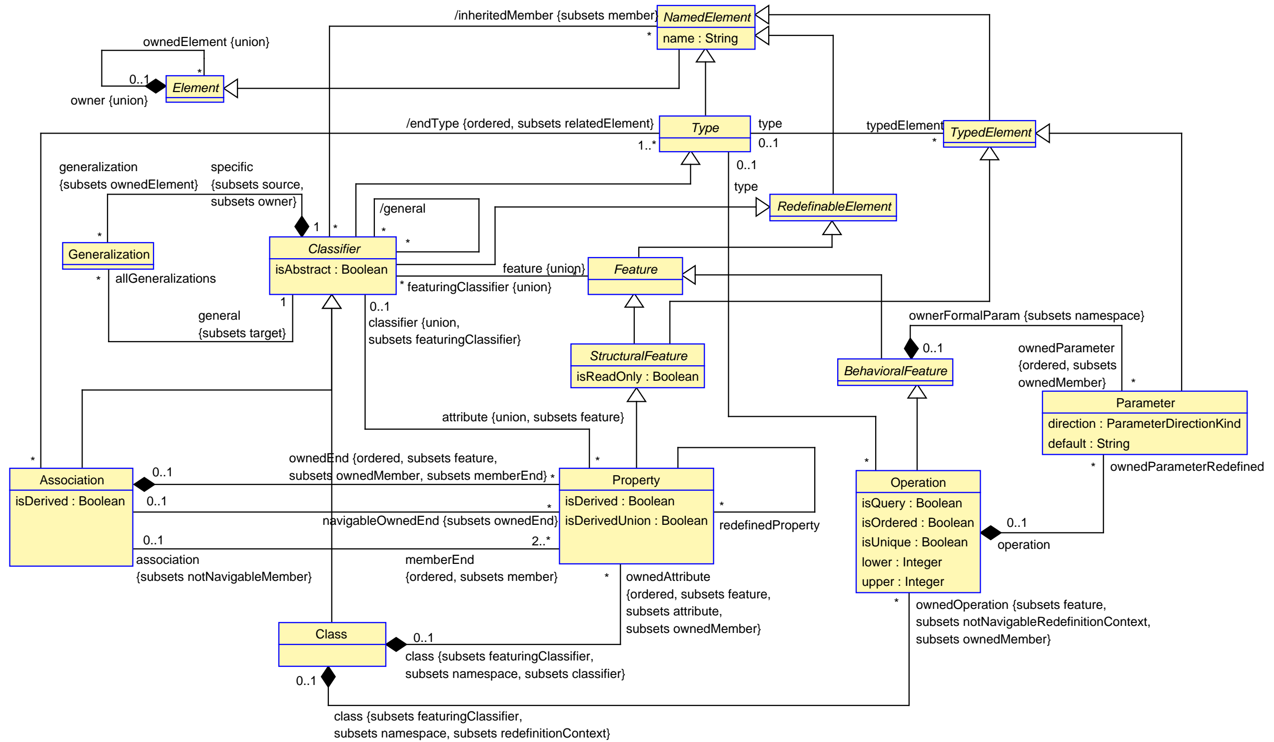

Figure 2. Central UML metamodel elements.

loaded in USE in order to generate the *metamodel instantiation* representing the user model.

B. Metrics Measurement

In this section, we will present how the *metamodel* is exploited for metrics measurement. The metrics are defined separately from the metamodel as operations of two newly added classes, one for class scope metrics and one for model/package scope metrics. These two classes are encapsulated in a new package, named `Metrics`, at the same level of the *metamodel*, i.e., M2 level.

1. Class scope metrics: In order to measure class scope metrics, e.g., NOC, NMO, DIT from Table I, first a class, named `ClassMetrics`, has been added to the `Metrics` package. This class is connected to the *metaclass* `Class` by an association as shown in Fig. 3. Therefore, from a class, one can access the metrics through the role name of this navigable association, i.e., `metrics`. Following this approach and in order to measure metrics for other elements of the *metamodel*, one can extend this option by creating a corresponding metric class in the Metrics package and an association between the created metric class and the element to be measured. Each metric is declared as an operation of the class `ClassMetrics` and the body is defined as an OCL expression (relying on the class diagram in Fig. 2).

These operations can access the *metamodel* in order to collect data for metrics calculation through the navigable association to the *metaclass* `Class`. One additional step must be performed in order to make these metrics ready for use. For each class, one *metaobject* of the corresponding *metaclass* `ClassMetrics` and a link are automatically added to the M1 level. Consequently, for every class in the *user model*, a pair of *metaobjects* (class, classmetrics) and a link between them are now available for metrics handling.

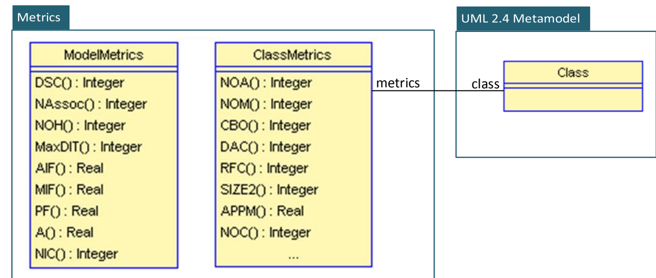

Figure 3. Metrics package and relationship to UML 2.4 metamodel.

To demonstrate the idea, we take the DAC (Data Abstraction Coupling) metric as an example. This metric is defined in [16] as follows:

Definition 1: Data Abstraction Coupling (DAC) is the number of attributes in a class having another class as their type.

Note that that from the original definition, DAC does not include the attributes that are inherited from super classes, only the attributes declared within the class itself are considered. The listing below shows the OCL expression definition of the

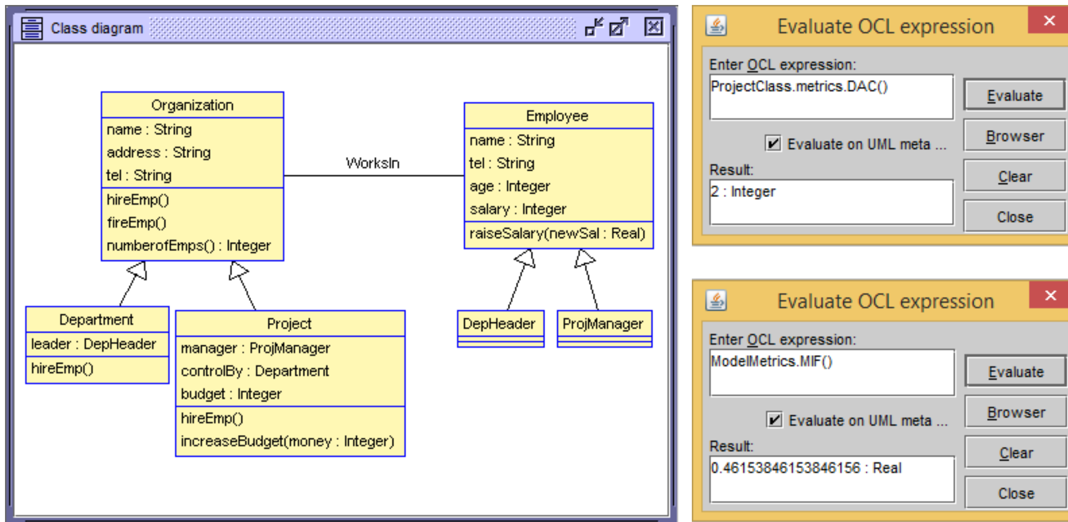

Figure 4. Example for metrics evaluation.

DAC metric in the context of the metaclass `ClassMetrics` as an operation.

```
DAC(): Integer =
    userdefinedTypeAttributes()→size()
```

where `userdefinedTypeAttributes()` is an auxiliary function that selects an OrderedSet of user-defined type attributes of the corresponding class. We have defined a list of auxiliary functions and these functions can be called within the definition of a metric.

```
userdefinedTypeAttributes(): OrderedSet(Property) =
    self.class.ownedAttribute
      →select(att|att.type.oclIsTypeOf(Class))
```

Within our approach, retrieving a metric of a class from the user class diagram can be done by the following expression template: <*Name of class*>.**metrics**.<*Name of the metric*> This is a natural way of retrieving a class metric, because a metric can be seen as a property of a class. Fig. 4 presents an example of a UML class metric measurement.

The left part of Fig. 4 is a simple class diagram containing major features of an object-oriented design, such as classes, attributes, operations, inheritance, and polymorphism. In the right upper part, the expression to retrieve the DAC() metric of the class `Project`, i.e., `ProjectClass.metrics.DAC()`, and the value, i.e., 2, are shown. The value 2 is caused by two attributes with user-defined type in the class `Project`, i.e., *manager* and *controlBy*. Note that Fig. 4 is a screenshot from the tool USE since our approach has been successfully integrated into USE. The value of the DAC metric can be used to analyze and predict the external quality of the software product to be implemented. DAC is a coupling metric: the higher the value of DAC, the less independent the class. Therefore, the DAC metric should not be too high, in order to improve the modularity and encapsulation of the design. When the coupling of a class is high, changing other classes is much more sensitive to the class under consideration, and therefore, the cost for maintenance in later phases is increased.

2. Model scope metrics: As can be seen in Fig. 3, all the model scope metrics are encapsulated in an isolated metaclass, i.e., `ModelMetrics`, in the Metrics package. Each metric is defined as an operation of the class `ModelMetrics` to measure the overall quality in the context of a class model, such as metrics for complexity, coupling, inheritance, polymorphism, or size. They are also formulated as OCL expressions at the meta level, and, within expressions, the pre-defined class scope metrics and the auxiliary functions can be called as well. In the following, an example of the definition and the use of a class model scope metric is presented, i.e., the MIF metric. The MIF metric is defined as follows [5]:

Definition 2: The Method Inheritance Factor (MIF) is defined as a ratio between the total number of inherited methods in all classes of the system under consideration and the total number of available methods (locally defined within the class and inherited from other classes) for all classes.

$$MIF = \frac{\sum_{i=1}^{TC} M_i(C_i)}{\sum_{i=1}^{TC} M_a(C_i)}$$

where:

- TC = total number of classes
- $M_i(C_i)$ = number of inherited methods in class C_i
- $M_a(C_i) = M_d(C_i)+M_i(C_i)$ = number of locally defined methods + number of inherited methods in class C_i

The following OCL expression is the formal definition of the MIF metric. It is defined as the body of an operation of the class *ModelMetrics*.

```
MIF(): Real =
    (Class.allInstances()→collect(c|c.metrics.NMI())→sum())/
    (Class.allInstances()→collect(c|c.metrics.NOM() +
          c.metrics.NMI())→sum())
```

In the above expression, two class scope metrics, i.e., NMI and NOM (see Table I), are called to yield the number of inherited methods and the number of locally defined methods in a class, respectively. The definition of these two class metrics can be found in [22]. The right lower part of Fig. 4 shows how the MIF metric can be achieved with our approach. The expression `ModelMetrics.MIF()` give us the corresponding value of the MIF metric of the class diagram on the left, i.e., 0.4615. This metric can be used to measure the level of reuse. A high value indicates a high level of reuse. According to [5] the value should not be too high. They proposed the value of the MIF metric should have an upper bound and should be smaller than 0.7 resp. 0.8.

3. Definition of new metrics: It could be that designers might find it difficult to define new metrics as OCL expressions over the UML metamodel on their own, and it could be an error-prone task for them. To overcome this drawback, we offer an interactive process for developing new metrics thanks to the availability of the *metamodel*, the *metamodel instantiation*, and the option for interactive OCL expression evaluation on the *metamodel* (as shown in the right-hand side of Fig. 4). A new metric can be the technical realization of what is typically called a "design smell". In particular, one can take an already defined metric as a template, and then modify the corresponding OCL expression (for example, one could change `Class` to `AssociationClass` in the definition of the DAC metric). The new metric, i.e, the new OCL expression over the UML metamodel, can be checked for syntax and tested on the generated *metamodel instantiation* of the current *user model*.

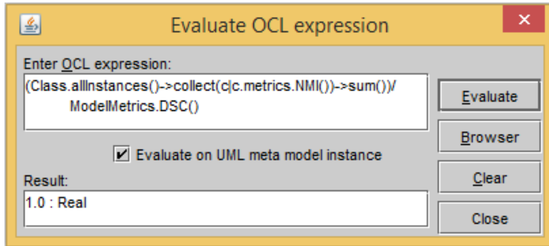

Figure 5. Defining a new metric.

This process can be iterated until a desired new metric has been developed. Finally, the successful new metric can be added to the Metrics package as an operation of the class `ClassMetrics` or the class `ModelMetrics` after giving it a name and then this metric is ready to be used in the future.

For example, if one wants to define a new metric, namely MIF1, that calculates the average number of the inherited methods per class in a *user model*. It can be done by taking the OCL expression definition of the MIF metric as a template, and then change it by replacing the later part by `ModelMetrics.DSC()` to get the total number of classes in the user model. This new OCL expression can be tested on the user model, e.g., the model in Fig. 4, as shown in Fig. 5.

IV. Model Quality Assessment with Metrics

As discussed in Sect. II, metric measurement can be used as an early indicator for software quality. Beside that, metrics can be exploited for design assessment or evaluation by setting different *thresholds* for different metrics. For example, one could set the possible maximum value of the DAC metric as 4, i.e., every class in the model should not have more than 4 attributes that have another class as their type. Or setting the value of the MaxDIT metric of the overall design between 1 and 5 means the model should have at least one inheritance tree longer than 1 and no inheritance tree longer than 5. Actually, the proposed metric threshold values might come from empirical studies (some of them have been indicated in the survey in [9]) or can be set by an experienced chief designer based on the requirement of a project. This assessment could not only help designers to control the quality of their work, but could help them to detect problems in the model as well. In this section we present an approach that supports designers (a) in the definition of a list of metrics, both class scope and model scope metrics, (b) in the specification of upper and lower thresholds, and (c) in the automatic evaluation of the model along the stated threshold settings.

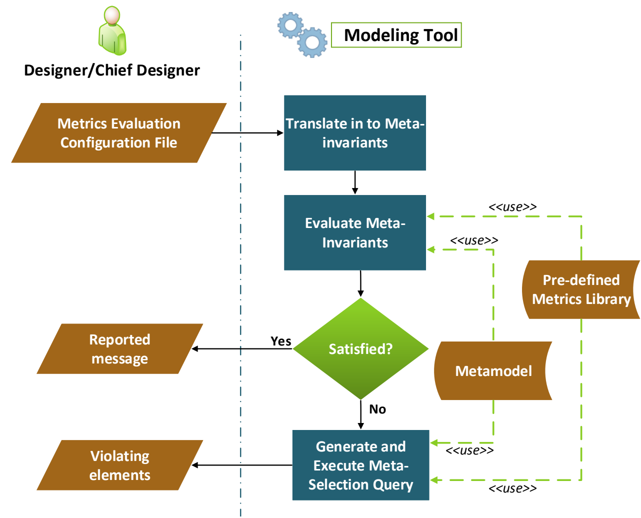

Figure 6. Workflow of model quality assessment with metrics.

Fig. 6 shows the workflow of the model quality assessment using a metric interval configuration. As can be seen from the workflow, the input of the assessment is a *metric evaluation configuration*. This configuration contains a set of desired metric threshold settings. These settings should be stated by an experienced chief designer. Each setting must contain the following information: metric scope, name of the metric as well as the lower and upper thresholds. Because the values of metrics are always either integer or real, the value of the lower and upper thresholds must be integer or real as well. The following listing is an example of a metric evaluation configuration, and it will be utilized for assessing the models with the combination of the metrics values.

```
0 NOM 4 15
0 DIT -1 5
0 NOA -1 10
0 NOC -1 4
1 DSC 5 20
1 MIF 0.2 0.8
```

Please note that, each line in this configuration contains the setting for one metric. In each line, from the left to the right, the corresponding values of the metric scope, the name of the metric, the lower threshold and upper threshold are stated. The first value 0 indicates that the corresponding metric is a class scope metric. The -1 value for lower or upper threshold settings means the corresponding metric is only upper bounded or lower bounded, respectively.

In the next step, the metric settings will be translated into a set of *meta-invariants*, one meta-invariant for each metric setting. We use the notion 'meta-invariant' because they are formulated on the metamodel (M2) and their evaluation is applied on the level of the user model (M1) (this rule is also applied to the later introduced type of query in the workflow, i.e., *meta-selection query*). The translation is based on the following strategy:

- **For class scope metrics**: we have to check the setting on all classes in the model. To achieve that, we propose the following template for evaluating the meta-invariant:

  ```
  Class.allInstances()→forAll( c|
    <the metric condition for class c>)
  ```

  where the metric condition is an expression for the comparison between the metric value and the lower and upper thresholds. If the lower or upper threshold value is missing (set as -1), the metric condition expression will not contain the comparison expression of the missing value. In the condition, the corresponding metric operation of class *c* is called. For example, the translation of the first setting in the above configuration file example, i.e., `0 NOM 4 15`, gives us the following meta-invariant:

  ```
  Class.allInstances()→forAll(c|
    4 <= c.metrics.NOM() and c.metrics.NOM() <= 15)
  ```

- **For model scope metrics**: because the context of these metrics is the complete model, the translated meta-invariants are simply basic comparison expressions between the metric value and the lower and upper thresholds. For example, the following listing is the meta-invariant which is translated from the last setting in the above configuration example, i.e., `1 MIF 0.2 0.8`

  ```
  0.2 <= ModelMetrics.MIF() and ModelMetrics.MIF() <= 0.8
  ```

Next, these meta-invariants must be evaluated on the metamodel level. This step can be performed by a supporting tool, since many modeling tools now support parsing and evaluating OCL expressions, for example, the tool USE. The result of this step is either an indication for satisfaction or violation of the metric threshold settings (meta-invariants), i.e., *true* or *false*. In case the evaluation for a class scope metric is unsatisfied (the evaluation is *false*), the designer may want to know which elements (classes) violate the setting. To find the violating classes in the model, we introduce a template for auto-generating a corresponding meta-selection query. The result of executing this meta-query is a set of classes that violate the threshold setting. The template for this kind of query is as follows:

```
Class.allInstances()→select(c |
  not <the metric condition for class c>)
```

The metric condition expression is constructed as mentioned above. For example, if the evaluation of the setting `0 NOM 4 15` shows the result *false*, the following OCL meta-query is automatically generated in order to find the violating classes, i.e., classes that have a number of local methods less than 4 or more than 15:

```
Class.allInstances()→select(c|
  not (4 <= c.metrics.NOM() and c.metrics.NOM() <= 15))
```

In order to show how the above proposed approach for model quality assessment with metrics can be successfully integrated into a modeling tool, Fig. 7 presents a simple example of applying this approach in the tool USE.

The class diagram of the model under consideration is presented on the left. Following the process shown in Fig. 6, using the metric configuration which is presented at the beginning of this section, the settings in the configuration are first translated into meta-invariants and then evaluated within USE. The right upper window shows the evaluation result. In particular, only the setting of the NOC metric, i.e., `NOC -1, 4`, is false. To explore which elements cause the NOC metric setting to be unsatisfied, the last step of the process is performed by double clicking on the row of the NOC metric setting. As a result, the violating class, i.e., the *Employee* class, is found and presented in the right lower part of Fig. 7. The *Employee* class has five direct sub-classes, which is out of the bounds of the NOC metric threshold setting.

V. Related Work

Class model metrics definition with OCL: In [23] an approach for object-oriented design metrics definition on the UML metamodel with OCL was introduced for the first time. The authors defined metrics as post-conditions of additional operations in the UML 1.3 metamodel. Continuing this idea, the work in [24] has made an extension by decoupling the metric definitions from the metamodel, which had been upgraded to the UML 2.0 metamodel, into a separate metrics package. The MOVA modeling tool, introduced in [25] provides a metamodel-based metrics measurement facility. In this work, the idea of an automatically generated metamodel instantiation for metrics definition was presented. Another approach [10] presented the SQUAM framework as a tool-supported method for metrics definition. Within this framework, 26 metrics for UML 2.2 class diagrams have been developed as additional operations of the metaclass *Class*. Our approach for metrics definition with OCL introduced here has utilized these works with extensions and improvements indicated below.

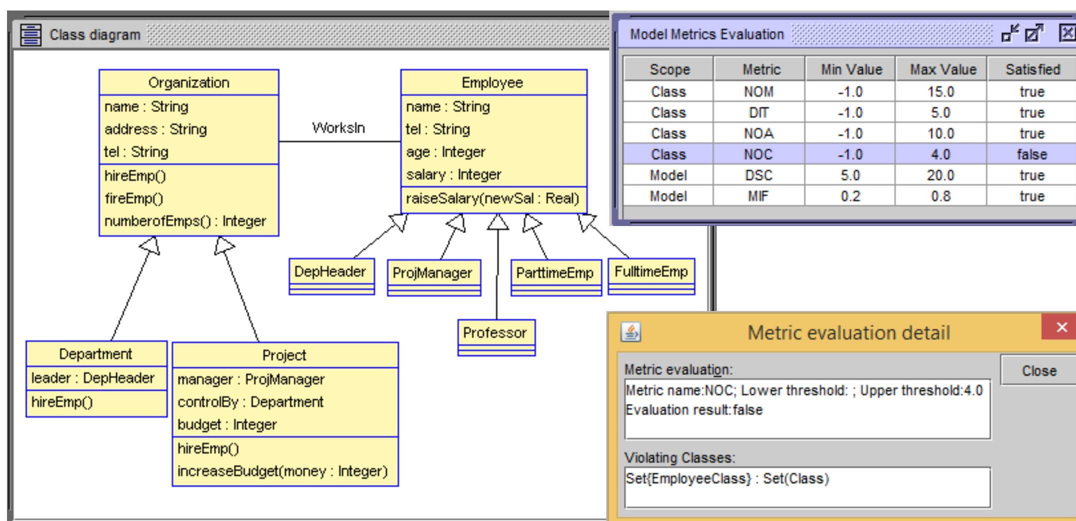

Figure 7. Example of model quality assessment with metrics.

- We have upgraded to the full UML 2.4 metamodel (with 69 classes and 99 associations).
- We offer an interactive process for developing new metrics thanks to the availability of the *metamodel*, the *metamodel instantiation*, and OCL expression evaluation on the *metamodel*. Moreover, designers can use the newly defined metrics immediately for measuring or assessing the model, for example, by applying our proposed assessment method.
- With our approach, one can achieve metrics similar to a property of the class itself. This is much more natural and straightforward for metrics treatment, since a metric can be considered as a property of a class.

Model quality assessment: The work in [26] has presented a unified method for the definition of UML class diagram quality properties, such as syntactic issues, best practices and naming issues. Another approach [27] was using the *mmSpec* language integrated into the tool *metaBest* to specify and check quality properties on UML models. In this work, the authors introduced some metric-related properties. In [28], Francesco et al. introduced an approach for quality assessment of modeling artifacts (metamodels, models, model transformations) by supporting hierarchical quality model definition using OCL and evaluate modeling artifacts based on metric measurements. Our basic idea of utilizing the UML metamodel for evaluating quality properties defined by OCL was introduced in [18]. However, one drawback of that work is the requirement of the basic knowledge of OCL and on the UML metamodel for quality properties definition. A common drawback of above approaches is the requirement of a modeling or constraint languages, e.g., OCL or mmSpec for the quality properties or attributes specification. With the assessment method proposed in this contribution, however, this drawback can be overcome. By simply providing a metric threshold configuration, designers will get the metrics evaluation result and the violating elements in case of unsatisfied thresholds. Our proposed assessment process can be smoothly integrated into any modeling tool which support OCL expression parsing and evaluation.

Metric definition using other languages: Besides OCL, several different languages have been used as the formal languages for model metrics definition. XQuery [29] and SQL [30] are examples from the academic context. Several tools have been developed for specification and calculation model metrics. EMF Metrics [31] is a prototype Eclipse plug-in working with models-based ideas in the Eclipse Modeling Framework. Within MagicDraw[1], UML metrics are measured by a Java hard-coded component, and the SDMetrics[2] tool works with models stored in the XML Metadata Interchange (XMI) format, and metrics are defined in the form of XML-based specifications. In contrast, our approach supports OCL, and we offer an interactive process for metrics definition (for developers with OCL expertise) or a threshold, template-based process (for developers with minimal or no OCL expertise).

VI. Conclusion and Future Work

In this paper, we have presented contributions on metric definitions and their application for internal quality assessment of models. We proposed a method to define UML metrics with OCL within a three level metamodeling approach. We have added within a three level tool the UML 2.4 metamodel to the topmost level and an automatically generated metamodel instantiation reflecting the user model in the middle level. This made metric definitions possible. The metrics have been formulated in OCL as the operations of the classes in a new and separate package in the topmost level. To show the feasibility of our approach, we have formulated a number of selected metrics from the literature and integrated them into the tool USE. A further contribution of our work was

[1]MagicDraw — http://www.magicdraw.com/
[2]SDMetrics — UML,http://www.sdmetrics.com/

to propose a complete process for model quality assessment with pre-defined metrics. In an assessment process, developers can achieve information about the quality of their design based on a metrics evaluation configuration, and they can detect problems in their model. Detailed OCL expertise is not required when a model is checked on the basis of such a configuration.

Future work can be done in various directions. First of all, we intend to formulate more metrics from the literature as well as newly defined ones within our proposed metrics definition method. Improving the usability aspect of the model assessment process will a task as well. For instance, a functionality showing all pre-defined metrics together with recommended upper and lower threshold settings collected from the studies in the literature would be a good solution for the configuration file. Another promising direction for future work would be to develop a prediction system for external software properties starting from internal quality indicators, i.e., metrics measurement. For instance, a decision making technique, e.g., the AHP [32] method for calculating the weights of external quality properties based on the measured metrics values and expert experience, could be integrated into our approach. Last but not least, larger case studies must give feedback on the applicability of the approach.

References

[1] Object Management Group – OMG, *Unified Modeling Language Specification, version 2.4.1*, 2011. [Online]. Available: http://www.omg.org/spec/UML/2.4.1

[2] J. Cabot and M. Gogolla, "Object constraint language (OCL): A definitive guide," in *Formal Methods for Model-Driven Engineering*, ser. Lecture Notes in Computer Science, M. Bernardo, V. Cortellessa, and A. Pierantonio, Eds. Springer Berlin Heidelberg, 2012, vol. 7320, pp. 58–90.

[3] S. R. Chidamber and C. F. Kemerer, "A metrics suite for object oriented design," *IEEE Trans. Softw. Eng.*, vol. 20, no. 6, pp. 476–493, Jun. 1994.

[4] M. Lorenz and J. Kidd, *Object-oriented Software Metrics: A Practical Guide*. Prentice-Hall, Inc., 1994.

[5] F. B. e. Abreu and W. Melo, "Evaluating the impact of object-oriented design on software quality," in *Proceedings of the 3rd International Symposium on Software Metrics: From Measurement to Empirical Results*, ser. METRICS '96. Washington, DC, USA: IEEE Computer Society, 1996, pp. 90–.

[6] L. Briand, P. Devanbu, and W. Melo, "An investigation into coupling measures for c++," in *Proceedings of the 19th International Conference on Software Engineering*, ser. ICSE '97. New York, NY, USA: ACM, 1997, pp. 412–421.

[7] R. Harrison, S. Counsell, and R. Nithi, "Coupling metrics for object-oriented design," in *Proceedings Fifth International Software Metrics Symposium. Metrics (Cat. No.98TB100262)*, Nov 1998, pp. 150–157.

[8] M. Genero, *Defining and validating metrics for conceptual models (Ph.D. thesis)*. University of Castilla-La Mancha, 01 2002.

[9] M. Genero, M. Piattini, and C. Caleron, "A survey of metrics for UML class diagrams," *Journal of Object Technology*, vol. 4, pp. 59–92, 2005.

[10] J. Chimiak-Opoka, "Measuring UML models using metrics defined in OCL within the SQUAM framework," in *Model Driven Engineering Languages and Systems, 14th International Conference, MODELS 2011, Wellington, New Zealand, October 16-21, 2011. Proceedings*, 2011, pp. 47–61.

[11] M. Gogolla, F. Büttner, and M. Richters, "USE: A UML-based specification environment for validating UML and OCL," *Sci. Comput. Program.*, vol. 69, no. 1-3, pp. 27–34, 2007.

[12] M. Gogolla and F. Hilken, "Model Validation and Verification Options in a Contemporary UML and OCL Analysis Tool," in *Proc. Modellierung (MODELLIERUNG'2016)*, A. Oberweis and R. Reussner, Eds. GI, LNI 254, 2016, pp. 203–218.

[13] ISO/IEC, *ISO/IEC 9126. Software engineering – Product quality*. ISO/IEC, 2001.

[14] J. Bansiya and C. G. Davis, "A hierarchical model for object-oriented design quality assessment," *IEEE Trans. Softw. Eng.*, vol. 28, no. 1, pp. 4–17, Jan. 2002.

[15] B.Henderson-sellers, *Object-Oriented Metrics, Measures of Complexity*. Prentice Hall, 1996.

[16] W. Li and S. Henry, "Object-oriented metrics that predict maintainability," *J. Syst. Softw.*, vol. 23, no. 2, pp. 111–122, Nov. 1993. [Online]. Available: http://dx.doi.org/10.1016/0164-1212(93)90077-B

[17] R. C. Martin, *Agile Software Development: Principles, Patterns, and Practices*. Upper Saddle River, NJ, USA: Prentice Hall PTR, 2003.

[18] K.-H. Doan and M. Gogolla, "Extending a UML and OCL tool for meta-modeling: Applications towards model quality assessment," in *Modellierung 2018*, 2018, pp. 205–220.

[19] Object Management Group – OMG, *OMG Meta Object Facility (MOF) Core Specification, version 2.4.1*, 2013. [Online]. Available: http://www.omg.org/spec/MOF/2.4.1

[20] Object Management Group, *OMG Unified Modeling Language(OMG UML), Superstructure, version 2.4.1*, 2011. [Online]. Available: http://www.omg.org/spec/UML/2.4.1/Superstructure

[21] F. Büttner and M. Gogolla, *Modular Embedding of the Object Constraint Language into a Programming Language*. Springer Berlin Heidelberg, 2011, pp. 124–139.

[22] K.-H. Doan and M. Gogolla, "Addendum to Evaluating UML Models by Applying Metrics within Metamodels," University of Bremen, Tech. Rep., 2018. [Online]. Available: http://www.db.informatik.uni-bremen.de/publications/intern/UMLModelsMetrics.pdf

[23] A. L. Baroni, S. Braz, F. B. e Abreu, and N. L. Portugal, "Using ocl to formalize object-oriented design metrics definitions," in *In Proc. of QAOOSE'2002, Malaga*. Springer-Verlag, 2002.

[24] J. A. Mcquillan and J. F. Power, "A definition of the chidamber and kemerer metrics suite for the unified modeling language," National University of Ireland, Tech. Rep., 2006.

[25] M. Clavel, M. Egea, and V. T. D. Silva, "Model metrication in mova: A metamodel-based approach using ocl," 2007.

[26] D. Aguilera, C. Gómez, and A. Olivé, "A Method for the Definition and Treatment of Conceptual Schema Quality Issues," in *Proc. 31st Int. Conf. ER 2012*, 2012, pp. 501–514.

[27] J. J. López-Fernández, E. Guerra, and J. de Lara, "Assessing the Quality of Meta-Models," in *Proc. 11th Workshop MoDeVVa@MODELS 2014.*, 2014, pp. 3–12.

[28] F. Basciani, J. D. Rocco, D. D. Ruscio, L. Iovino, and A. Pierantonio, "A customizable approach for the automated quality assessment of modelling artifacts," in *10th International Conference on the Quality of Information and Communications Technology, QUATIC 2016, Lisbon, Portugal, September 6-9, 2016*, 2016, pp. 88–93. [Online]. Available: http://doi.ieeecomputersociety.org/10.1109/QUATIC.2016.025

[29] M. A. M. El-wakil, A. El-Bastawisi, M. B. Riad, and A. A. Fahmy, "A novel approach to formalize object - oriented design metrics," in *Conf. Evaluation and Assessment in Software Eng*, 2005.

[30] T. J. Harmer and F. G. Wilkie, "An extensible metrics extraction environment for object-oriented programming languages," in *2nd IEEE International Workshop on Source Code Analysis and Manipulation (SCAM 2002), 1 October 2002, Montreal, Canada*, 2002, pp. 26–35. [Online]. Available: https://doi.org/10.1109/SCAM.2002.1134102

[31] T. Arendt, P. Stepien, and G. Taentzer, "Emf metrics: Specification and calculation of model metrics within the eclipse modeling framework," in *BENEVOL*, 2010.

[32] T. Saaty, "Decision making with the analytic hierarchy process," *International Journal of Services Sciences*, vol. 1, pp. 83–98, 01 2008.