Generation of Formal Model Metrics for MOF based Domain Specific Languages

Marcus Engelhardt, Christian Hein, Tom Ritter, Michael Wagner

Fraunhofer FOKUS, Kaiserin-Augusta-Allee 31,
10589 Berlin, Germany
{Marcus.Engelhardt, Christian.Hein, Tom.Ritter,
Michael.Wagner}@fokus.fraunhofer.de

Abstract. The assessment of quality in a software development process is vital for the quality of the final system. A number of approaches exist, which can be used to determine such quality properties. In a model-driven development process models are the primary artifacts. Novel technologies are needed in order to assess the quality of those artifacts. Often, the Object Constraint Language (OCL) is used to formulate model metrics and to compute them automatically afterwards. This paper describes an approach for the generation of model metrics expressed as OCL statements based on a set of generic rules. These rules can be applied on any domain specific modeling languages for creating a basic set of metrics which can be tailored for the specific needs of a development process. The paper also briefly describes a prototype of a tool for the generation, computation, and management of these model metrics by using the Software Metrics Meta-model - SMM.

Keywords: model metrics, OCL, SMM

1 Introduction

Quality of software has become essential to Software Engineering so that increasingly more resources are provided for tasks dealing with quality assurance in software development processes. In particular, the early and continuous quality assessment can provide quantitative indicators for model quality and help to locate structural problems. But measuring certain properties of software is a hard, time- and resource-consuming task since the tool support for automated quality measurement is still lacking and hence a lot of manual work is required.

In Model Driven Engineering (MDE) the model is the primary artifact of the development process. The quality of the involved models has a significant influence on the quality of the final software. Due to the central relevance of a model, the quality requirements for it increase. While numerous quality characteristics for code artifacts have been identified and standardized in various quality models in recent

years, the definition of appropriate quality criteria for models is still not well established.

An often used means to determine software quality are metrics. Applied to several artifacts of the software during its whole life cycle, they can produce comparable evaluations of these artifacts as a basis for later assessments of quality properties. Numerous metrics defined on code level can be found in literature. In terms of model driven development, a number of approaches to define metrics, which are useful to determine the quality of models, have been proposed in the meantime. A number of them are referenced in section 2.

Due to the fact that there is no standard terminology for defining metrics, a grand challenge is to find an appropriate mechanism to define them. While the first software metrics had been mostly defined using natural language, which may result in ambiguity, others have been expressed using a formalism with a defined semantics. However, the latter requires some formal background for understanding. In our approach we use the Object Constraint Language (OCL) [6] for metrics definition which is widely accepted as an interesting balance between formality and understandability.

Most of the model metrics proposed up to now aim to measure quality properties related to architectural design. Therefore, many of these metrics are defined on the Unified Modeling Language (UML) meta-model in context of UML classes, etc., but are not usable at a lower level. Thus, a tool which defines and deals with metrics for domain specific languages (DSL) written in OCL, in particular with capabilities to generate these, is still missing. In this paper we present an approach to automatically generate domain specific metrics for Meta Object Facility (MOF) based meta-models and a concept to manage and compute them on models which conforms to these meta-models.

The paper is organized as follows. In section 2 we briefly summarize related work of model metric definitions and tools for applying metrics to models. In section 3 our approach to derive formal model metrics from meta-models will be described. Thereby, we briefly introduce the Software Metrics Meta-model SMM by the Object Management Group (OMG) and describe the main phases of the tool's metrics management and computation concept. Furthermore, an overview over some metric generation rules and an example for each of them are given including a description. In section 4 we describe a prototype implementation of the above mentioned approach called Metrino [21]. Finally, in section 5, we draw conclusions of our work.

2 Related Work

A large number of object oriented metrics is available in literature. A broad overview and comparison object oriented design model metrics are given in [12]. Several approaches can be found in literature addressing the problem of ambiguous metric definitions. A tool which calculates metrics for object-oriented languages is presented in [1]. In this approach, the metrics are written as SQL queries over a relational database schema which serves as the meta-model for the definition of metrics. In [2]

the XML Query Language (XQuery) [18] is used to define metrics based on the XMI serializations of meta-models. Another approach [3] proposes a formal model for object-oriented design called ODEM (Object-oriented Design Model) as a foundation to formally define metrics dealing with object-oriented design.

Baroni et al. [4] were the first authors who proposed the use of OCL (Object Constraint Language) to formalize object oriented metrics. They described the MOOD metrics based on a meta-model called GOODLY using OCL. Inspired of well known suites of metrics like MOOD, MOOD2, MOOSE, EMOOSE AND QMOOD, they have developed a library called FLAME (Formal Library for Aiding Metrics Extraction) [5] which contains several object oriented design metrics upon the UML 1.3 meta-model formally expressed with OCL invariants.

While numerous approaches for metric tools dealing with formal metrics on code level like EMBER [13] has been proposed in recent years, the set of metric tools on model level is comparatively small. One approach for the latter is the MOVA tool [14]. Beside some facilities to draw UML class and object diagrams and formulate OCL constraints to precise models, the tool provides functionality to manually create and compute OCL based metrics for user models in a proprietary environment. For the metric application, the tool internally maps a user model to instances of the MOVA meta-model which itself is a subset of the UML meta-model. However, the metrics the tool can deal with, are only applicable to the user models, but not on instances of them. Thus, it is not able to provide the ability to define specific metrics based on the domain the user-defined meta-model describes.

[15] proposes a set of tools named MOODKIT G2 for the extraction of MOOD design metrics from various OOD formalism such as code of OO programming languages like C++ and Java and models expressed in modeling languages like UML. For each input type, a specific parser implementation is needed. MOODKIT G2 uses a textural object oriented design language named GOODLY as a meta-model for the metric definition.

Furthermore, Borland Together [20] is a representative of a so called COTS tool which supports metric computation as well. Basically, it is a modeling tool that follows the MDA approach by supporting the essential technologies such as UML modeling, OCL and QVT. In terms of quality assurance, it utilizes the approach of Baroni et al. [4]. The metrics and modeling guidelines, called audits, are also specified as OCL expressions. A standard set of metrics, applicable to UML2 models, is provided. The metric set can be adapted and extended. However, they are limited to UML2 models and no metric generation for DSLs is available.

There are not many tools available which take domain specific languages into account. One of these tools to mention in this context is DMML (Defining Metrics at the Meta Level) [17]. They picked up the approach of [4] and decoupled its metric definitions from the underlying meta-model by defining a separate metrics package containing a single class. Each metric is specified as an operation in this class using an OCL body expression. In order to prove the concept, they implemented the Chidamber and Kemerer metric suite upon the UML 2.0 meta-model for the DMML tool. However, the tool is relatively hard to extend with other meta-models since the user manually has to define a transformation/mapping of the meta-model to a XML

schema the tool understands. In addition, the user manually has to create the metrics as OCL queries in a separate file according to the metric names he has to specify when loading the corresponding meta-model. This is necessary, because the tool first creates the metrics package extension to the meta-model needed as the basis for the generation of the corresponding Java classes for the metric computation on instances of a model conform to the meta-model. Although the DMML tool is - with some effort - usable to define metrics for domain specific languages, it does not provide any concept for an approach for the generation of metrics for DSLs.

Another relevant approach is the one presented in [20]. It deals with the visual specification of measurements (metrics) and refactorings for any Domain Specific Visual Language (DSVL). The approach is based on graph pattern matching. Visual patterns expressed in a DSVL called SLAMMER are used to specify relevant elements for a measurement and refactoring type. Within a meta-modeling tool called AToM, these patterns can be applied to a DSVL in order to generate a modeling environment for the language providing corresponding concrete measurements and refactorings. In addition, the generated environment allows to trigger refactorings when a metric' threshold is reached or exceeded. Though this approach is dedicated to the same problem, it works in a complete different way since it does not OCL but graphical patterns.

3 Automatic Generation of Metrics

3.1 Meta-model for Metric Definition

Our approach uses the Software Metrics Meta-model (SMM) for the definition of metrics and their computational results. The SMM specification [11] distinguishes between **measures** as the evaluation process of particular quality aspects of software artifacts and **measurements** which can be interpreted as the results of those processes. For consistency reasons, we will follow this naming convention.

Currently being available in beta status, SMM is a specification consolidated by the OMG for an extensible meta-model that primarily should establish an interchange of measurements over existing software artifacts. Those artifacts could be source code or - more interesting in the context of this paper – models, as well. SMM contains meta-model classes for numerous types of measures and their measurements including a set of contextual information.

As mentioned above, the meta-model specifies several types of measures and measurements for different outcome values of the evaluation processes (measures). The latter could assign either numeric values of a domain with a pre-defined ordering relation or numeric values representing ratios (e.g. percentages). In addition, the SMM provides appropriate classes for the mapping of values of a particular interval to a related symbol. In terms of SMM, these types of measures are called *Ranking*. Symbolic values like "good", "satisfying" and "bad" can be considered as such a *Ranking*.

The two basic measure types are *DimensionalMeasure* for numerical evaluation result values and *Ranking* for correspondent symbolic result values. Furthermore, the meta-model specifies three subtypes of the *DimensionalMeasure* class. One of them is *DirectMeasure* whose result value refers to the return value of a given operation stated in the corresponding property of the class. Another type is the *BinaryMeasure* which associates two base measures and accumulates their evaluation results using a binary function (*functor*). The last to mention subtype of *DimensionalMeasure* is the *CollectiveMeasure* class. Measures of this type are usable for model elements which aggregate other elements (children). Applied to such a container, a *CollectiveMeasure* itself applies its related base measure to each aggregated element to obtain a set of base measurements. Afterwards, these values are combined to the overall value for the measurement of the *CollectiveMeasure* itself using a particular accumulator like sum, minimum, maximum, etc. The set of available accumulators is extensible by providing corresponding specializations of the *CollectiveMeasure* class.

The last to mention SMM measure type in this paper is the *Counting* class which is a subclass of *DirectMeasure.* The given operation of a *Counting* measure acts as a recognizer function and has to return either 0 or 1 based upon recognizing the measurand.

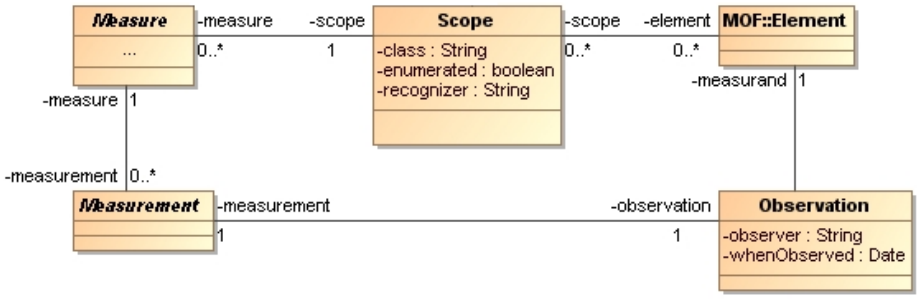

Fig. 1. Fundamental approach of the SMM meta-model

The fundamental approach of the SMM meta-model is shown in Fig. 1. Each measure has a *Scope* which determines the set of possible elements the measure can be applied to. This set can be constraint either by mentioning a class name each element in the scope should be an instance of, the explicit enumeration of member elements (a set of *MOF::elements*) or the reference to a boolean operation. Referred to as the *recognizer*, the latter provides a boolean value for each element of the examined model which determines whether the particular element should be a member of the scope or not.

As already mentioned, each measure produces a *Measurement* as result of the evaluation process. For each measure type the SMM specifies a corresponding subtype of the *Measurement* class, e.g. for *DirectMeasure* the *DirectMeasurement* class with a property *value* holding the result value of the related operation. Each measurement has an association to an object of type *Observation* which stores various

contextual information related to the measurement, such as the time of evaluation, the responsible tool, etc.

3.2 Model Metrics Generation Rules

The definition of model measures using OCL expressions has been done extensively for UML models as outlined in section 2. Following these approaches, we also use OCL in our concept to describe measures formally. However, in contrast to many other available solutions, we target on a general approach which is capable of dealing with measures on any model conform to the MOF meta-model. In practical environment we have identified the necessity to provide measures for DSLs whose significance increase.

The definition of domain specific measures could be a very time consuming task, so that a generative approach would be desirable. Therefore, beside the possibility to define custom concrete measures for a domain specific model, we provide concepts to automatically generate domain specific measures based on a set of generic **rules**. Fig. 2 shows the correlation between the different relevant terms.

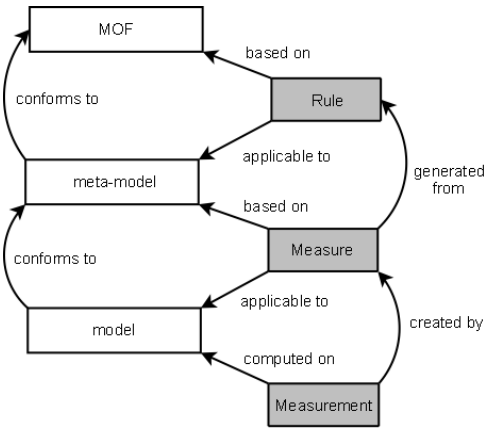

Fig. 2. Correlations between models, rules and measures

The generation rules themselves are formulated as OCL expressions upon the Meta Object Facility (MOF) [7] meta-model. As part of the Eclipse Modeling Framework [8], the Ecore meta-model is a famous and widely used implementation of the essential part of MOF (EMOF) and thereby practically suitable to serve as the meta-model for the OCL expressions used in generation rule definitions. The evaluation of a rule's OCL expression when applying the rule to a MOF conform meta-model results in a set of OCL tuples. OCL has been chosen in order to support an excepted standard rather than introducing an own proprietary format representing a SMM Measure. Each tuple provides measure specific data, e.g. its scope and its generated

name, which are required to generate the corresponding SMM Measure for the meta-model in context. The type of the generated measures is always identical to the measure type of the corresponding rule. In case of a rule of type *DirectMeasure*, the data additionally has to contain a value for the measure's operation property value which itself is an OCL expression. A detailed explanation of these correlations is given for one of the generation rules presented in the remainder of this section.

Regarding the measures defined in the currently available metric sets, we encountered basic patterns and common aims of particular groups of metrics. These patterns lead to the definition of numerous generation rules which allow the automatic derivation of domain specific measures depending on the meta-model, the rules are applied to. The generated measures are almost ready to run on models (instances of the meta-model) they are derived from. The only thing, the user has to add in order to make a measure runnable, is its threshold value which naturally depends on the acceptable value for the particular quality property being measured. The measures generated on the basis of this rules can be tailored for the specific requirements of a particular development process. In fact, these measures are able to provide a new kind of model quality assessment since they work on an arbitrary meta-model and assess quality aspects within a particular domain.

In the following sections we describe the set of generation rules we identified so far. For each rule we shortly summarize the pattern which upon the rule is based and we illustrate the rule by applying them on a common example. Each application of a generation rule results in a set of measures presented as well. Due to space limitation we only present the complete specification of the first rule and of the resulting measures. For the specifications of all rules detected so far we refer to the appendix.

The sample meta-model used for the following rule examples is based on a meta-model of the Eclipse help [9] and illustrated in Fig. 3. It is a simplified version of a library concept space.

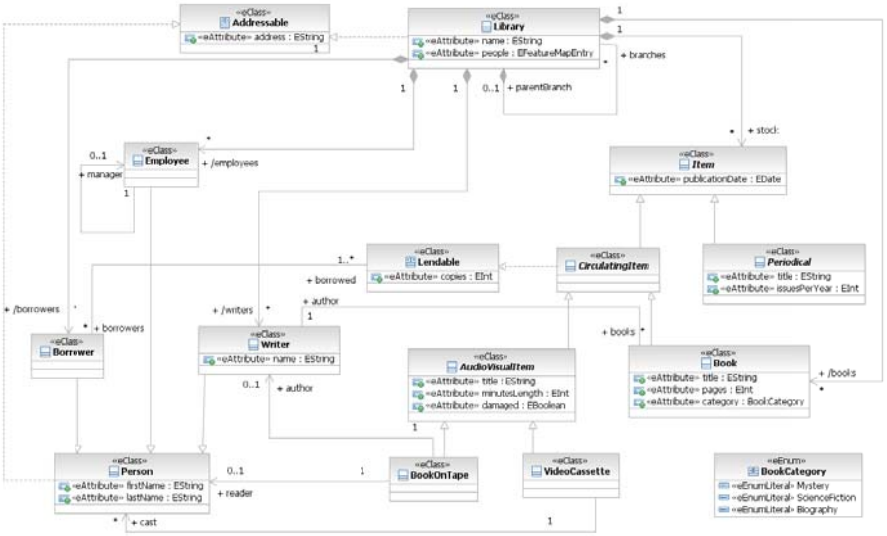

Fig. 3. Sample meta-model of a library

3.2.1 Partitioning based on enumeration typed class property

Description. The rule is applicable to classes in the analyzed meta-model having one or more attributes with an enum type. Since an enumeration defines a data type with a finite domain, it is possible to partition the set of instances of such a class based on the enumeration's literals. The rule generates a specific *Collective* measure with sum accumulator and a corresponding *Counting* measure as base measure for each enumeration literal which aims to count the instances of a matching class grouped by the particular values of the enumerated type. Therefore, the number of the generated measures is equal to the number of the literals of the examined enumeration type.
As mentioned above, rules in our approach are defined upon the MOF meta-model. Since we use EMF in our prototype, a rule is defined with elements of the Ecore meta-model. The model classes and associations, the rule uses, are shown in Fig. 4.

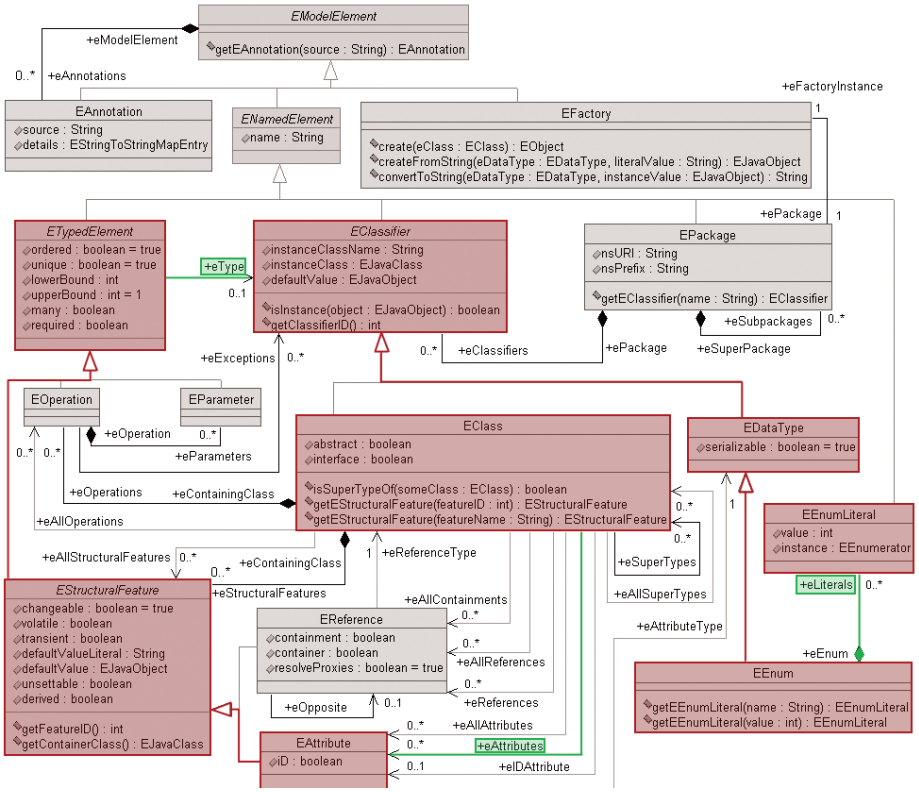

Fig. 4. Model elements in Ecore meta-model used by the rule

The OCL operation property of the base measure which has elements of type *EClass* in scope is formally specified as:

```
self.eAttributes->select(a | a.eType.oclIsKindOf(EEnum))
      ->collect(a | a.eType.oclAsType(EEnum).eLiterals->collect(l | Tuple {
      scope = self, nameFragments = Sequence {'NoOf',
      a.eType.oclAsType(EEnum).getEEnumLiteral(l.value).literal.upperFirst(),
      self.name.upperFirst()}, operationFragments = Sequence{
      'self.', a.name,
      ' = ', a.eType.name, '::', a.eType.oclAsType(EEnum).getEEnumLiteral(l.value)
      .literal}}))
```

This rule uses an OCL helper operation “upperFirst” to transform the first letter of a string into the corresponding uppercase letter (see appendix).

Example. The *Book* class of the library model has an attribute *category* of the enumerated type *BookCategory*. This enumeration type contains three literals, namely *Mystery*, *ScienceFiction* and *Biography*. The generation process on the library model using this rule would result in the following three OCL tuples describing the resulting base measures:

```
Tuple{scope = Book, nameFragments = [NoOf, Mystery,
Books], operationFragments = [self., category,  = ,
BookCategory, ::, Mystery]}
Tuple{scope = Book, nameFragments = [NoOf,
ScienceFiction, Books], operationFragments = [self.,
category,  = , BookCategory, ::, ScienceFiction]}
Tuple{scope = Book, nameFragments = [NoOf, Biography,
Books], operationFragments = [self., category,  = ,
BookCategory, ::, Biography]}
```

The information provided by a tuple for a base measure are also used to generate the *Collective* measure for the model class having a containment reference to the class in scope of the corresponding base measure. In case of the first tuple the *Collective* measure *NoOfMysteryBooks* will be generated for the enumeration literal *Mystery*. This measure uses a generated base measure of type *Counting* whose operation property in OCL will be defined as follows:

```
self.category = BookCategory::Mystery
```

When computed on a snapshot (instance) model of the Library meta-model, the *Collective* measure *NoOfMysteryBooks* will apply its base measure to all *Book* instances contained in an instance of the *Library* class. For each Book whose category value is equal to the enumeration literal *Mystery*, the *Counting* base measure will return the value "1" and thereby increment the measurement value of the *Collective* measure.

3.2.2 Number of contained Elements

Description. The rule detects classes in a meta-model having containment references. For each match a specific measure is derived which aims to count the referenced elements of a detected containing element in an instance model of the meta-model.
Example. According to the example library meta-model the rule produces six measures for the Library class. These are:

- NoOfBorrowersLibrary
- NoOfEmployeesLibrary
- NoOfWritersLibrary
- NoOfBranchesLibrary
- NoOfStockLibrary
- NoOfBooksLibrary

3.2.3 Partitioning based on boolean class property

Description. The rule matches to meta-model classes which have at least one attribute of type Boolean. For each of these attributes the rule leads to the generation of a

model specific measure which counts the instances of a matching class whose value for the examined attribute is “true”.

Example. Applied to the example library meta-model, the rule generates three measures for the attribute *damaged* of the model class *AudioVisualItem*. First of all, the measure *NoOfDamagedAudioVisualItem* will be generated. Due to the fact that the classes *BookOnTape* and *VideoCassette* are subclasses of *AudioVisualItem*, the rule would additionally create the measures *NoOfDamagedBookOnTape* and *NoOfDamagedVideoCassette*, respectively.

3.2.4 Referential optionality

Description. The rule matches to model classes that are the origin of an association with a lower bound value equal to zero. For each match the rule causes the generation of a measure which counts the instances of a matching model class where the reference’s upperbound value is equal to zero.

Example. Applied to our example meta-model the rule generates the following measures:

- NoOfBookOnTapeWithoutAuthor
- NoOfBookOnTapeWithoutReader
- NoOfEmployeeWithoutManager
- NoOfLibraryWithoutParentBranch
- NoOfLibraryWithoutBranches
- NoOfLibraryWithoutBorrowers
- NoOfLibraryWithoutBooks
- NoOfLibraryWithoutStock
- NoOfLibraryWithoutEmployees
- NoOfLibraryWithoutWriters
- NoOfWriterWithoutBooks
- NoOfLendableWithoutBorrowers
- NoOfVideoCassetteWithoutCast

3.2.5 Depth of instance tree

Description. This rule matches to nested class structures in a meta-model. These hierarchies are modeled through recursive class associations or associations between classes joining the same inheritance hierarchy (composite), respectively.

This rule can be considered as a semantic equivalent to the Chidamber & Kemerer metric *Depth of inheritance tree (DIT)* shifted to a lower model level.

Example. An example for a nested structure, this rule matches to, can be found in the example meta-model. The Library class has a containment reference to the *Library* class itself. This association is meant to model a branch hierarchy of a library.

Applied to the library model, the rule would create a measure named *DepthInLibraryTree* which is useful to calculate the depth of each branch, e.g. instance of the *Library* class, in the library’s hierarchy of branches. In case of class

inheritance the name of the top level generalization class is used to compose the measure's name.

3.2.6 Number of children of an instance

Description. Like the rule *Depth of instance tree* this rule is applicable to nested class structures in a meta-model. For each match a specific measure will be generated which calculates the number of children of each class instance in the hierarchy.
Example. Based on the example library model, the rule would generate a measure for the *Library* class. This measure, which would be called *NoOfChildrenLibrary*, counts the number of branches of each particular *Library* class instance.

3.2.7 Referential existence dependencies between classes

Description. This rule is meant to generate measures which will detect referential existence dependencies between instances of model classes sharing an association. These dependencies can be found regarding the lower bound value of the association. If this value is greater than zero, the existence of the instances of a class at the opposite association end are logically dependent on the existence of instances of the class with the mentioned association end (having the lower bound value greater than zero). The number of minimal required class instances is determined by this lower bound value.

This rule assumes that instances which are associated with just so many instances, as the lower bound value suggests, have a greater significance in the model since the deletion of one of the associated instances will affect the deletion of the independent instance in order to maintain the model valid.
The rule generates measures for each model class having an association to another class with a lower bound value greater than zero for the opposite association end. A created measure counts the instances of the scope's class associating just as many instances of the other class as the lower bound value of the meant association end claims.
Example. In case of the example library meta-model, the rule would match to the association between the *Borrower* class and the interface *Lendable*. A borrower will only be captured in the model if he borrows at least one lendable item. This is modeled by the lower bound value "1" for the association end *borrowed*. Applied to the example meta-model, the rule would create the measure *NoOfBorrowersMinimalBorrowed* which will count those instances of the *Borrower* class that are associated exactly with only one instance of a class implementing the *Lendable* interface.

3.2.8 Instance (usage) ratio in inheritance structures

Description. The rule applies to inheritance structures in a meta-model. For each subclass in an inheritance hierarchy it creates a measure which calculates the ratio of

the number of its instances to the total number of the (polymorphic) instances of its super type.
Example. Focusing the inheritance tree of the library meta-model's class *Item*, the rule would generate the following six measures:

- RatioOfCirculatingItemToItem
- RatioOfPeriodicialToItem
- RatioOfAudioVisualItemToCirculatingItem
- RatioOfBooksToCirculatingItem
- RatioOfBookOnTapeToAudioVisualItem
- RatioOfVideoCassetteToAudioVisualItem

3.2.9 Aggregation of associated elements

Description. This rule matches to classes which have at least one multiplicity-many association. It creates measures which apply aggregate functions to the instances of such a class to assess the class-wide maximum, minimal, average, etc. value of referenced elements count.
Example. Regarding the association *employees* of the *Library* model the rule would create measures like:

- MaxEmployeesAllLibrary
- MinEmployeesAllLibrary
- AvgEmployeesAllLibrary

3.2.10 Aggregation based on numeric property

Description. This rule matches to classes which have at least one property with a numeric type. It generates measures which apply aggregate functions to the instances of such a class to assess the class-wide maximum, minimal, average, etc. value for the examined property.
Example. Referring to the *Book* class of the library meta-model this rule would lead to the generation of measures like:

- MaxPagesAllBook
- MinPagesAllBook
- AvgPagesAllBook

4 Tool Support - Metrino

In this section we introduce a prototype implementation of our approach for the generation of domain specific measures and its main operational phases. To be practically useful, the tool additionally provides the functionality to manage and compute generated or user-defined measures and to visualize their computational

results in appropriate charts. The application of the presented rule set could result in a huge number of metrics (measures). Not all of them are of the same importance and therefore an adequate and tool-supported metrics management is inevitable. The tool is implemented as a set of plugins for the Eclipse IDE and uses the OSLO library [10] for the evaluation of OCL expressions occurring in both, rules and measures.

Since rules and measures in our tool are defined using OCL query expressions, all values of the operations defined in the SMM meta-model, like the *recognizer* property of the *Scope* class and the *operation* property of a *DirectMeasure*, are OCL expressions which are evaluated by the OSLO OCL processor used in the prototype implementation. An example OCL expression is presented in section 3.2.1.

4.1 Operation phases

The Metrino tool basically operates in four phases: the rule management phase, the measure generation phase, the measure management phase and the measure evaluation phase. An overview of these phases is depicted in Fig. 5.

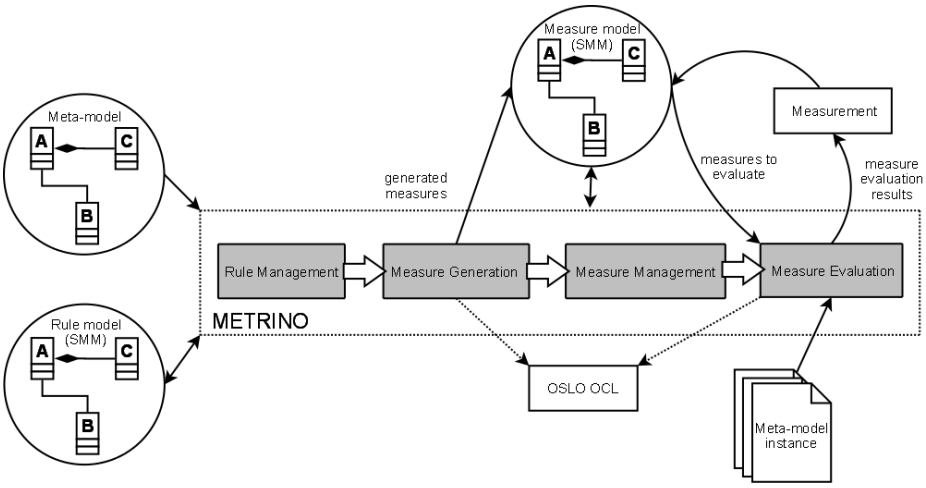

Fig. 5. The four phases of Metrino

In the rule management phase, the user can either define generation rules from the scratch or load an existing rule model (e.g. based on the rule set presented in section 3.2) and modify its contained rules. The rules defined in the rule model serve as input for the measure generation phase. In order to run the generation process, the user first has to load a meta-model, measures should be derived from, and then select the rules to use from a selection dialog (rule selection). The tool then applies each selected rule to the meta-model and generates a measure for each match. By default, the generated measures are grouped in SMM categories on the basis of the model types they are applicable for. The result of the measure generation is an exportable/savable measure model (SMM model) containing all the generated domain specific measures or even a

tailored subset of them. The measures are almost ready to be applied to an instance model. The only property, the user has to specify/modify, is the measure's threshold value in order to obtain feasible graphical reports of the measure's evaluation results (measurements).

Measure management phase: The generated measures can be modified freely. Of course, the user additionally can define custom measures for the model or drop existing ones. In addition, the user can group a particular set of measures to run at once, which assess the same quality aspect of the model, as a so called audit.

In the measure evaluation phase, the tool applies a set of measures, the user has selected before, to an instance of the model these measures have been generated for. Along with the evaluation time and some other contextual information, the resulting measurements are stored in the corresponding measure model.

It is important to mention that the four phases do not necessarily have to be run through starting from the first – the rule management – phase since the user can load a existing rule model or measure model. In this case, the rule management phase and the measure management phase are optional so that the user directly can continue with the measure generation and the measure evaluation, respectively.

The tool provides several charts for the graphical processing of evaluation results available in the measure model. Beside Kiviat graphs to display numerous measurements for an element at once and various other graph types, the tool is able to visualize the change of measurements over time since the date of each measurement is available in the measure model. Fig. 6 shows a screenshot of the prototype implementation depicting the main views of the tool and a Kiviat graph for some measurement results.

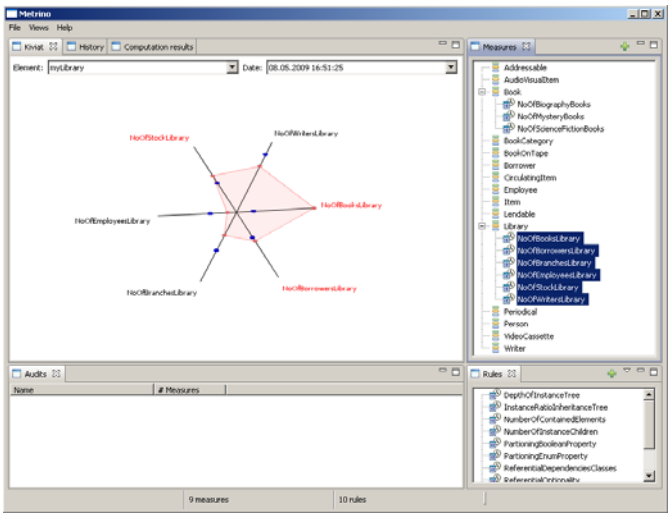

Fig. 6. Screenshot Metrino

5 Conclusion and Further Work

In this paper, we have identified the need for a metric tool which is able to deal with domain specific model metrics. Since the definition of metrics on this level could be very time consuming due to their uniqueness, an approach for the automatic generation of those measures would be desirable. As outlined in section 4, we were not able to find an approach or even (prototype) implementation which provides such a facility using OCL. In order to meet this deficiency, we have introduced a set of generic rules derived from patterns of model measures already available in literature. Furthermore, we have described the fundamentals of our approach for generating metrics for domain specific languages. Following the work in [5, 16, 17] we use OCL for the formal definition of both rules and metrics.

In order to facilitate our approach, we have developed a prototype which allows the generation and evaluation of model metrics for DSLs. Based on the SMM as the underlying meta-model for the definition of metrics and their computational results, the prototype implementation additionally offers some functionality to manage rules and measures as well as several facilities for the graphical processing of measure evaluation results and even its comparison over history.

In further steps, we plan to enhance our approach and the Metrino tool in different ways. First of all, we plan to extend the set of measure generation rules in order to provide a broader basis for the generation of domain specific measures. In addition, we intend to realize the measure generation process as a model-to-model transformation using Query View Transformation (QVT) [19]. Therewith, we will avoid the creation of an intermediate structural specification for the generated measures like the currently used OCL tuple sets a rule returns. Moreover, we plan to extend the Metrino tool with several features to create comprehensive reports for measure evaluation results in standard formats such as Microsoft Office document formats. Finally, we target on applying our approach to more industrial case studies in order to get more detailed feedback on the usability of our approach.

6 Acknowledgement

This research has been co-funded by the European Commission within the 6th Framework Programme project Modelplex contract number 034081 (cf. http://www.modelplex.org).

7 References

1. Wilkie, F.G., Harmer, T.J.: Tool Support for Measuring Complexity in Heterogeneous Object-Oriented Software. In: Proceedings of the International Conference on Software Maintenance (ICSM'02), 152 (2002)
2. El-Walkik, M. M., El-Bastawisi, A., Riad, M. B., Fahmy, A.A.: A novel approach to formalize object-oriented design metrics. In: Proceedings of Evaluation and Assessment in Software Engineering (EASE'05) (2005)
3. Reißing, R.: Towards a Model for Object-Oriented Design Measurement. In: Quantitative Approaches in Object-Oriented Software Engineering, (QAOOSE'01), 2001
4. Baroni, A.L., Braz, S., Brito e Abreu, F.: Using OCL to Formalize Object Oriented Design Metrics Definitions. In: Proceedings of ECOOP Workshop on Quantitative Approaches in Object-Oriented Software Engineering, Malaga, Spain (2002)
5. Baroni, A., Brito e Abreu, F.: A Formal Library for Aiding Metrics Extraction. In: 4th International Workshop on OO Reengineering, Darmstadt, Germany (2003)
6. OMG: Object Constraint Language. http://www.omg.org/docs/formal/06-05-01.pdf
7. OMG: Meta Object Facility (MOF) Core Specification. http://www.omg.org/spec/MOF/2.0
8. Eclipse Foundation: Eclipse Modeling Framework. http://www.eclipse.org/modeling/emf/
9. Eclipse Foundation: Eclipse documentation. http://help.eclipse.org/ganymede/topic/org.eclipse.emf.validation.doc/references/examples/exampleOverview.html
10. Fraunhofer Institute for Open Communication Systems: Open Source Library For OCL. http://oslo-project.berlios.de/
11. OMG: Software Metrics Meta-Model (SMM) 1.0 – Beta 1. http://www.omg.org/docs/ptc/09-03-03.pdf
12. El-Wakil, M., El Bastawissi, A., Boshra, M., Fahmy, A.: Object-Oriented Design Quality Models A Survey and Comparison. In: 2nd International Conference on Informatics and Systems (INFOS04), Cairo, Egypt, 2004
13. Wilkie, F.G., Harmer, T.J.: Tool Support for Measuring Complexity in Heterogenuous Object-Oriented Software. In: Proceedings of the International Conference on Software Maintenance (ICSM'02), Montreal, Canada (2002)
14. Clavel, M., Egea, M., Torres da Silva, V.: Model Metrication in MOVA: A Metamodel-based Approach using OCL. (2007)
15. Brito e Abreu, F., Ochoa, L., Goulão, M.: The GOODLY Design Language for MOOD Metrics Collection. (1997)
16. McQuillan, J.A., Power, J.F.: A definition of the Chidamber and Kemerer Metrics Suite for UML. Report NUIM-CS-TR2006-03, Department of Computer Science, National University of Ireland, Maynooth, Ireland (2006)
17. McQuillan, J.A., Power, J.F.: Towards the re-usability of software metric definitions at the meta level. In: European Conference on Object-Oriented Programming, Nantes, France (2006)
18. W3C: XQuery 1.0: An XML Query Language. http://www.w3.org/TR/xquery/
19. OMG: Meta Object Facility (MOF) 2.0 Query/View/Transformation Specification. http://www.omg.org/docs/formal/08-04-03.pdf
20. Guerra, E., De Lara, J., Díaz, P.: Visual Specification of Measurements and Redesigns for Domain Specific Visual Languages. Journal of Visual Languages and Computing Vol. 19(3), Elsevier (2008)
21. Metrino: A component of ModelBus. https://www.fokus.fraunhofer.de/de/motion/ueber_motion/technologien/metrino/index.html

8 Appendix

This appendix includes the formal specification of the metric generation rules we have identified so far and the OCL helper operations the prototype tool uses in these specifications.

8.1 Formal Specifications of Metric Generation Rules

This section lists the OCL specifications of the metric generation rules introduced in section 3.2.

8.1.1 Partitioning based on enumeration typed class property

Measure type:	Collective measure (accumulator: sum)
Base measure type:	Counting measure
Scope:	EClass

Base measure operation specification:

```
self.eAttributes->select(a | a.eType.oclIsKindOf(EEnum))
     ->collect(a | a.eType.oclAsType(EEnum).eLiterals->collect(l | Tuple {
     scope = self, nameFragments = Sequence {'NoOf',
     a.eType.oclAsType(EEnum).getEEnumLiteral(l.value).literal.upperFirst(),
     self.name.upperFirst()}, operationFragments = Sequence{
     'self.', a.name,
     ' = ', a.eType.name, '::', a.eType.oclAsType(EEnum).getEEnumLiteral(l.value)
     .literal}}))
```

8.1.2 Number of contained Elements

Measure type:	Direct measure
Scope:	EClass

Measure operation specification:

```
self.eAllContainments->collect(c | Tuple {scope = self, nameFragments = Sequence {
     'NoOf', c.name.upperFirst(), self.name.upperFirst() }, operationFragments =
     Sequence {'self.', c.name , '->size()'}})
```

8.1.3 Partitioning based on boolean class property

Measure type:	Collective measure (accumulator: sum)
Base measure type:	Counting measure
Scope:	EClass

Base measure operation specification:

```
self.eAttributes->select(a | a.eType.name = 'EBoolean')->collect(ba | Tuple {
       scope  = self, nameFragments = Sequence{'NoOf', ba.name.upperFirst(),
       self.name.upperFirst()}, operationFragments = Sequence{'self.', ba.name , '
       = true'}})
```

8.1.4 Referential optionality

Measure type: Collective measure (accumulator: sum)
Base measure type: Counting measure
Scope: EClass

Base measure operation specification:

```
self.eReferences->select(r | r.lowerBound = 0)->collect(rl | Tuple {
       scope  = self,  nameFragments = Sequence{'NoOf', self.name.upperFirst(),
       'Without', rl.name.upperFirst()}, operationFragments = Sequence{'self.',
       rl.name, '->size() = 1'}})
```

8.1.5 Depth of instance tree

Measure type: Direct measure
Scope: EClass

Measure operation specification:

```
self.eAllContainments->select(r | r.eOpposite->notEmpty()  and (r.eType.name =
       self.name or r.eType.oclAsType(EClass).eAllSuperTypes->exists(r2 | r2.name
       = self.name)))->collect(r3 | Tuple { measure = Tuple { scope = self,
       nameFragments = Sequence{'DepthIn', self.name.upperFirst() , 'Tree'},
       operationFragments = Sequence{'self.', r3.eOpposite.name, 'Depth()'}},
       helpers = Sequence {r3.eOpposite.name, 'Depth(): Integer = if self.',
       r3.eOpposite.name, '->isEmpty() then 0 else self.', r3.eOpposite.name, '.',
       r3.eOpposite.name, 'Depth() + 1 endif'}})
```

Each generated measure will make use of a special generated OCL helper operation (also defined in the rule specification) which is meant to determine the number of parent (container) elements of a particular element being measured in the scope of the generated measure.

8.1.6 Number of children of an instance

Measure type: Direct measure
Scope: EClass

Measure operation specification:

```
self.eAllContainments->select(r | r.eType.name = self.name or
       r.eType.oclAsType(EClass).eAllSuperTypes->exists(r2 | r2.name =
       self.name))->collect(r3 | Tuple { scope = self, nameFragments =
       Sequence{'NoOfChildren', self.name.upperFirst()}, operationFragments =
       Sequence{'self.', r3.name, '->size()'}})
```

8.1.7 Referential existence dependencies between classes

Measure type:	Collective measure (accumulator: sum)
Base measure type:	Counting measure
Scope:	EClass

Base measure operation specification:

```
self.eReferences->select(r1 | r1.lowerBound > 0)->collect(r2 | Tuple { scope  =
       self, nameFragments = Sequence {'NoOf', self.name.upperFirst() ,'Minimal',
       r2.name.upperFirst()}, operationFragments = Sequence{'self.', r2.name ,
       '->size() = ', r2.lowerBound.oclAsType(String)}})
```

8.1.8 Instance (usage) ratio in inheritance structures

Measure type:	Direct measure
Scope:	EPackage

Measure operation specification:

```
self.eClassifiers->select(e | e.oclIsKindOf(EClass))->reject(e |
       e.oclAsType(EClass).eSuperTypes->isEmpty())->collect(c1 |
       c1.oclAsType(EClass).eSuperTypes->collect(c2 | Tuple { scope  = self,
       nameFragments = Sequence {'RatioOf',
       c1.oclAsType(EClass).name.upperFirst(), 'To',
       c2.oclAsType(EClass).name.upperFirst()}, operationFragments = Sequence
       {c1.oclAsType(EClass).name, '.allInstances()->size() / ',
       c2.oclAsType(EClass).name, '.allInstances()->size()'}}))
```

8.1.9 Aggregation of associated elements

Measure type:	Collective measure (accumulator: sum, min, max, avg, etc.)
Base measure type:	Direct measure
Scope:	EClass

Base measure operation specification:

```
self.eReferences->select(r | r.upperBound > 1 or r.upperBound = -1)->collect(mr |
       Tuple {scope = mr.eContainer(), nameFragments = Sequence
         {mr.name.upperFirst() , self.name.upperFirst()}, operationFragments =
         Sequence {'self.', mr.name, '->size()'}})
```

8.1.10 Aggregation based on numeric property

Measure type:	Collective measure (accumulator: sum, min, max, avg, etc.)
Base measure type:	Direct measure
Scope:	EClass

Base measure operation specification:

```
self.eAttributes->select(a | a.eType.name = 'EInt' or a.eType.name = 'EReal')
        ->collect(na | Tuple {scope = self.eContainer(), nameFragments =
        Sequence{na.name.upperFirst(), self.name.upperFirst()},
        operationFragments = Sequence{'self.', na.name}})
```

8.2 Specifications of OCL helper operations used in rule definitions

This section lists the OCL helper operations that are used in the specifications of the metric generation rules.

8.2.1 upperFirst()

```
context String def: upperFirst() : String =
        let charsLower : Sequence(String) =
        Sequence{'a','b','c','d','e','f','g','h','i','j','k','l','m','n','o','p',
        'q','r','s','t','u','v','w','x','y','z'} in

        let charsUpper : Sequence(String) =
        Sequence{'A','B','C','D','E','F','G','H','I','J','K','L','M','N','O','P',
        'Q','R','S','T','U','V','W','X','Y','Z'} in

                if charsLower->includes(self.substring(1,1)) then
                        charsUpper->at(charsLower->indexOf(
                        self.substring(1,1))).concat(self.substring(2,
                        self.size()))
                else
                        self
                endif
```