Andreas Oberweis, Ralf Reussner (Hrsg.): Modellierung 2016,
Lecture Notes in Informatics (LNI), Gesellschaft für Informatik, Bonn 2016 29

Deriving Model Metrics from Meta Models

Nebras Nassar[1], Thorsten Arendt[1], Gabriele Taentzer[1]

Abstract: The use of model-based software development has become more and more popular because it aims to increase the quality of software development. Therefore, the number and the size of model instances are cumulatively growing and software quality and quality assurance consequently lead back to the quality and quality assurance of the involved models. For model quality assurance, several quality aspects can be checked by the use of dedicated metrics. However, when using a domain specific modeling language, the manual creation of metrics for each specific domain is a repetitive and tedious process. In this paper, we present an approach to derive basic model metrics for any given modeling language by defining metric patterns typed by the corresponding meta-meta model. We discuss several concrete patterns and present an Eclipse-based tool which automates the process of basic model metrics derivation, generation, and calculation.

Keywords: Model metrics, metric patterns, quality assurance, Eclipse Modeling Framework.

1 Introduction

The paradigm of model-based software development (MBSD) has become more and more popular since it promises an increase in the efficiency and quality of software development. In this paradigm, models play an increasingly important role and become primary artifacts in the software development process. In particular, this is true for model-driven software development (MDD) [SVC06] where models are used directly for automatic code and test generation, respectively. Furthermore, the use of domain specific modeling languages (DSMLs) [BCW12] is a promising trend in modern software development processes to overcome the drawbacks concerned with the universality and the broad scope of general-purpose languages like the Unified Modeling Language (UML) [UML].

Since software models play an increasingly important role, software quality and quality assurance consequently lead back to the quality and quality assurance of the involved models. In [Ar11, Ar14], we introduced a structured syntax-oriented process for quality assurance of software models that can be adapted to project-specific and domain-specific needs according to a dedicated quality model (QM). The approach concentrates on quality aspects to be checked on the abstract model syntax and is based on quality assurance techniques model metrics, smells, and refactorings well-known from literature.

Metrics are useful to obtain quantitative information about software development processes and artifacts. Metrics for measuring the success of modeling and analysis has always been a challenge, especially in the area of enterprise modeling where very large models are in practical use. They can be used to analyze model quality, especially to find anomalies in

[1] Philipps-Universität Marburg, {nassarn,arendt,taentzer}@informatik.uni-marburg.de

models, and to estimate project costs. To measure quality aspects (e.g., model complexity) of DSMLs, basic model metrics are needed such as: total number of model elements of a specific type, number and average number of model elements of a specific type owned by a model element, number of incoming (outgoing) links of a specific type to (from) a model element, and average number of incoming (outgoing) links of a specific type per model element within the entire model. More really domain-specific metrics can be composed from already defined ones.

For evaluating quality issues we adopt the Goal-Question-Metrics approach (GQM) that is widely used and has been well established in practice [Va02]. Figure 1 illustrates the steps of the GQM process described as follows: 1. To measure the quality of a model, the first step is to define a measurement goal such as *model comprehensibility* [MDN09]. 2. Questions should be defined to support data interpretation towards a measurement goal. For example, the following question could be derived since a complex model is hard to understand: *How complex is a model wrt. the number of its elements?* 3. Metrics should be defined that help to provide all the quantitative information to answer the questions in a satisfactory way. A way to measure complexity is to use the metric *cyclomatic complexity* defined as *number of links* $-$ *number of elements* $+$ *2* [Mc76, Mc82] wrt. a control flow graph. To define model metrics for a given domain, the following tasks should be done: (a) find out (basic and complex) metrics needed to collect the related quantitative information and define them, (b) find and derive the corresponding domain metrics wrt. the definitions by analyzing and understanding the given domain structure, (c) identify the specification of each derived metric, and (d) find out which domain information is needed for specifying and implementing the derived metrics. 4. Once this information is identified, the corresponding code and artifacts of metrics calculation have to be developed for each derived metrics. 5. Implemented metrics can be used to analyze specific models. 6. After the defined metrics have been measured, sufficient information should be available to answer the questions. A cyclomatic complexity of 10, for example, points to a pretty complex control flow which might be hard to understand.

Fig. 1: GQM process

Considering a DSML being either a completely new language or a changed one due to evolution steps [HW14], its language and tool designers have to offer enough tool support for convenient domain-specific modeling processes. Specifying and defining metrics for a DSML by hand is time-consuming and error-prone. Although the definition and implementation of metrics cannot be automated completely, new ways are interesting to reduce the manual effort as much as possible. An approach and corresponding tool support for automatically deriving basic metrics from any given modeling language definition seems to be promising. Automatically deriving basic metrics, the effort of specifying and implementing model metrics would be reduced to composing basic metrics in a suitable way. Considering again the GQM process in Figure 1, Steps 1, 2 and 6 would remain manually, while 3 - 5 would be largely tool-supported.

The contributions of this paper are the followings:

- An approach to derive basic model metrics for any given modeling language (DSML or GPML [2]) being defined by a meta-model. The approach is especially useful for DSMLs to support the development of useful model metrics.
- An Eclipse-based tool to automate the definition of basic metrics for any given domain. The outcome of the tool is a high-level tool specification as an Eclipse plug-in which comprises the specification of the derived metrics, and code generation for metrics calculation, reporting and composition. (Steps 3 - 5 in Figure 1). The generated plug-in is used as input to EMF Refactor [Ref] for metrics calculation and composition.

Generating basic domain-specific metrics from a given meta-model helps us to concentrate on those metrics that really demand domain-specific knowledge. Our generation approach does not stop at trivial metrics but can also incorporate more complex ones such as the cyclomatic complexity and LCOM (Lack of cohesion of methods), or the definition of model queries. We illustrate our approach by using a DSML for simple Petri nets and discuss selected metrics patterns which are useful to derive basic metrics for this domain. Moreover, we present an implementation of the approach based on the Eclipse Modeling Framework (EMF) [EMF, St08] and EMF Refactor. To demonstrate the applicability and usefulness of the approach, we present an example application of this implementation on the UML class model domain.

The paper is structured as follows: The following section presents an example modeling scenario motivating our work. In Section 3, we present our approach for deriving basic model metrics in detail. Selected metrics patterns and their application on the example scenario are discussed in Section 4. After presenting the Eclipse-based tool prototype in Section 5, we demonstrate the applicability and usefulness of the approach in Section 6. Section 7 compares to related work and Section 8 concludes the paper.

2 Running Example: Basic Metrics for Petri Nets

In this section, we motivate our work by using an example Petri net scenario. We first describe the corresponding Petri net meta model. Thereafter, we discuss several basic metrics which can be used to analyze concrete Petri nets, i.e., instance models of this meta model.

Figure 2 shows the meta model of a Petri net language. A Petri net is composed of several places and transitions. Arcs between places and transitions are explicit: *PTArc* and *TPArc* are respectively representing place-to-transition arcs and transition-to-place ones. Arcs are annotated with weight. Each transition has at least one input place and one output place. Places can have an arbitrary number of incoming and outgoing arcs. In order to model dynamic aspects, places need to be marked with tokens. Figure 3 shows an example Petri net instance modeling some specific dynamic behavior. The Petri net consists of four

[2] Here, *GPML* refers to any general-purpose modelling language.

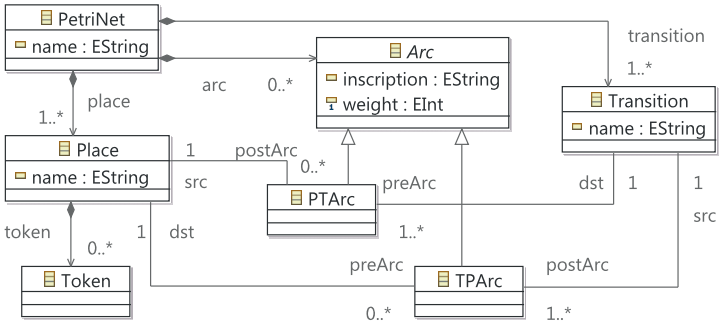

Fig. 2: Petri net meta model

places (P1 to P4), two transitions (T1 and T2) and altogether seven arcs connecting these elements. Please note that we omit arc weights and inscriptions for simplicity reasons.

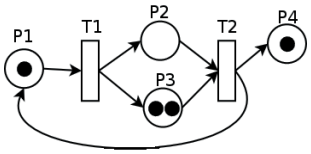

Fig. 3: Example Petri net instance

We present an example of the GQM process applied on the Petri net DSML as follows: Let the quality goal be the comprehensibility of Petri net models, one derived question is: *How complex is a model wrt. the number of its elements (places and transitions)?* A suitable metric to answer this question is the `cyclomatic complexity` defined as *number of tparcs and ptarcs − number of places and transitions + 2*. Applying this metric to the given Petri net instance (in Figure 3), the result is 3. Hence, there are only 3 independent paths and the given model is easy to understand and maintain.

To define metrics for analyzing models such as the *cyclomatic complexity*, some basic metrics are useful [Ch95, SJM92], e.g., metrics which simply count elements of specific types (like *number of transitions in the Petri net*, *number of places in the Petri net* and *number of arcs in the Petri net*). These basic metrics may be composed using arithmetic expressions to define the desired metrics.

When analyzing basic metrics such as the ones which calculate average values (like *average number of outgoing (or incoming) arcs of all transitions in the Petri net*), we observe that the structure of how they are specified is generic to some extent. The information needed can be obtained from three classes in the meta model (in our case *PetriNet*, *Transition*, and *TPArc*) where the first two classes are connected by a containment reference (*transition*) and the latter two classes are connected by a non-containment reference (*postArc*). In the following, we identify recurring patterns in meta models that are useful to derive basic model metrics.

3 Approach

Considering tool support for domain specific modeling, we are faced with the dilemma that we have to set up the tooling for metric calculation for each domain specific modeling language, even for basic metrics. And if the DSML has changed due to some evolution steps (e.g., the evolution of the Petri net meta model as described in [HW14]), its tooling has to be adapted. Therefore, we address the following research problem throughout this paper:

> How can the information, stored in a meta model, be used to automate the process of creating tool support for calculating basic metrics on corresponding instance models?

As we have seen in the preceding section, some basic metrics are defined for the Petri net domain such as *Number of transitions in the Petri net* and *Number of tokens in the Petri net*. These metrics could be abstractly described as *Number of instances of type X in an instance of type Y*. Moreover, we observe that several kinds of metrics can be derived by the same (or at least similar) abstract description. This abstract description can be used to specify basic metrics for any given domain by using the concrete domain data like the name of the corresponding domain element, e.g., *Transition*, *PetriNet*, and *Token*.

So, our approach is to design several metrics patterns (i.e., structural descriptions) which can represent the abstract description structure of several kinds of domain-specific metrics. Domains are usually defined by a domain-specific language, more specifically by a meta model. Therefore, the metrics patterns have to be typed by the meta-meta model so that the patterns can be applied over any given domain (meta-model) to find and derive the correspondences (the pattern matches). These correspondences within a specific domain hold the concrete domain data needed to define, specify and generate basic domain-specific metrics. These metrics are executable on instance models. In the following, we present the process for defining a new metrics pattern and deriving basic metrics from this pattern for a concrete meta model.

1. First, we design a pattern over the meta-meta model. This pattern consists, e.g., of two nodes and a containment relation in between as shown in Figure 4.

Fig. 4: A simple example of a metrics pattern

2. Then, the pattern can be applied to a concrete domain in order to find and retrieve all the pattern matches (correspondences) whose structures are instances of the pattern structure. For the Petri net example presented in Section 2, the following pattern matches are found:

   - Node *PetriNet*, Node *Place* and a containment relation named *place*.

- Node *PetriNet*, Node *Transition* and a containment relation named *transition*.
- Node *Place*, Node *Token* and a containment relation named *token*.

3. From each pattern match we now derive one or more basic model metrics. The following Petri net metrics can be derived from the matches described above:
   - Number of *places* in a selected *Petri net*.
   - Number of *transitions* in a selected *Petri net*.
   - Number of *tokens* in a selected *place*.

   Using the data of the pattern matches we can define and specify the model metrics for the given domain. Thereafter, an existing tool for metrics calculation on DSML instance models may be extended.

Our approach helps to easily produce the ”boilerplate” information of metrics specification. Having basic domain-specific metrics at hand, metrics and queries which require real domain-specific knowledge can be specified on top of those.

To sum up, metrics patterns are designed independent of concrete DSMLs. These patterns can be used to find and derive basic domain-specific metrics. Using the data of the retrieved pattern matches, we can derive basic domain-specific metrics and specify them in an appropriate query language like OCL. Furthermore, the corresponding code can be also generated in order to calculate metric values of concrete instance model. Figure 5 illustrates our approach.

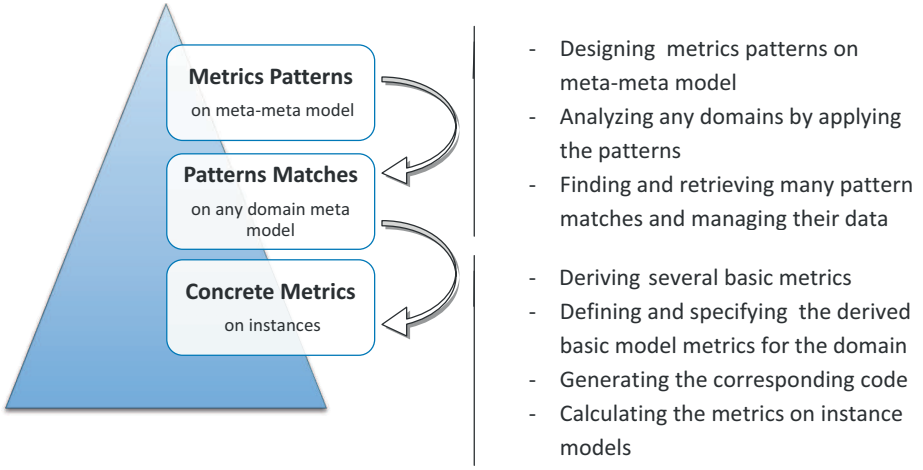

Fig. 5: The general approach to derive basic model metrics

Specifying thresholds to interpret metric results, however, is out of scope of this work due to individual interpretation opportunities depending on the modeling language, the modeling purpose, and the quality aspect considered by the measurement, respectively [Ar14].

4 Metrics Patterns

We developed 20 metrics patterns typed by Ecore, the meta-meta model of EMF [EMF, St08] representing the reference implementation of the Essential MOF (EMOF) standard for defining meta models using simple concepts [MOF]. The patterns are divided into four groups depending on the number of the *EReference* nodes in each pattern. In this section, we firstly present two selected patterns in detail: the simplest pattern and a more complex one. Finally, we give an overview about the remaining patterns.

4.1 Selected Metrics Patterns

Each metrics pattern can be used to describe the abstract structure of at least one kind of metrics. In summary, the 20 metrics patterns can derive altogether 42 kinds of basic model metrics. In the following descriptions of two selected patterns, the term *concrete pattern* refers to an instance of the basic metrics pattern.

Example for a simple metrics pattern

Figure 6 shows a simple concrete pattern (in concrete syntax) matched to the Petri net domain. It simply consists of one single class (*Place*) and can be used to derive and specify metric *Number of places in the model*. Now, our goal is to find all concrete patterns (matches) which have the same structure so that we can derive metrics that have the same abstract form. The pattern can be any class such as *Place* as shown in Figure 6.

Fig. 6: A node-related pattern

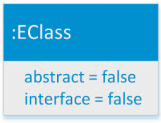

Fig. 7: Metrics pattern

Figure 7 shows the corresponding pattern (in abstract syntax) typed by Ecore. It consists of only one node of type *standard EClass* in order to represent non-abstract classes. [3] Applying this pattern to the Petri net domain, several concrete pattern matches exist as, e.g., *Transition*, *Token*, *PTArc* and *TPArc*. Each match can be used to specify a concrete metric. Example metrics for the Petri net domain are: number of *transitions* in the *model*, number of *tokens* in the *model*, number of *place-to-transition arcs* in the *model*, and number of *transition-to-place arcs* in the *model*.

As a result, this pattern can be used to find and derive domain metrics which have the following abstract description:

Number of all instances of type *X* in the model.

Here, *X* represents the name of any matched class from any given domain with respect to the applied pattern.

[3] Here, *standard EClass* means, that meta attributes *abstract* and *interface* are set to *false*.

Example for a more complex metrics pattern

Figure 8 shows a more elaborated concrete pattern (in concrete syntax) matched to the Petri net domain. It consists of class *PetriNet* connected by a containment relation named *transition* to class *Transition* which is in turn connected to class *TPArc* (the referenced class) by a non-containment relation named *postArc*.

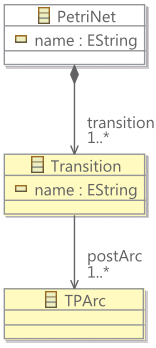

Fig. 8: An edge-related pattern

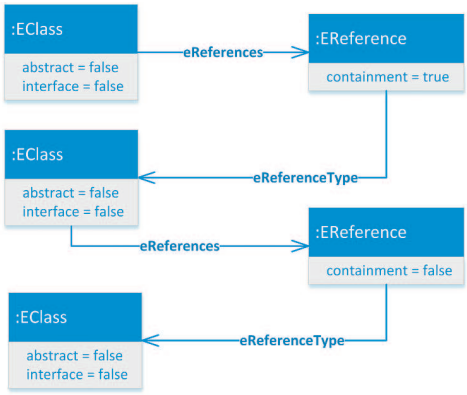

Fig. 9: Metrics pattern

This concrete pattern can be used to derive and specify the following concrete metrics:

- Number of all *outgoing arcs* of all *transitions* in a selected *Petri net*.
- Average number of *outgoing arcs* of all *transitions* in a selected *Petri net*.

Again, we want to find all concrete pattern matches for a given domain. Any concrete pattern must consist of three classes with one containment relation and one non-containment relation between these classes. Figure 9 shows the corresponding pattern (in abstract syntax), again typed by Ecore. The pattern consists of altogether five nodes. The nodes on the left are of type *standard EClass* (see above). Here, the top-left node is used to infer a container class, the middle-left node is used to infer a contained class , and the bottom-left node is used to infer a referenced class. The other nodes are of type *EReference*. Here, the containment attribute of the top-right node is set to *true* whereas the containment attribute of the other one is set to *false*. This means that the top-right node is used to infer the containment relation between the corresponding matched classes whereas the bottom-right node is used to infer the non-containment one.
As a result, this pattern can be used to find and derive metrics which have the following two abstract descriptions:

Number of all instances of type *X* referenced by all instances of type *Y* in a selected instance of type *Z*.
Average number of all instances of type *X* which are referenced by all instances of type *Y* in a selected instance of type *Z*.

Here, *X* represents the name of the matched referenced class, *Y* is the name of the matched contained class and *Z* is the name of the matched container class.

4.2 Metrics Pattern Groups

As mentioned above, we developed 20 metrics patterns which are separated into four groups depending on the number of *EReference* nodes in each pattern. In the following, we present an overview about the patterns in these groups and the kind of metrics which can be derived.

Group "Single node pattern" The first group contains only one pattern which consists of only one node of the type *EClass* to derive the atomic metrics (see Figure 7).

Group "One-edge patterns" The second group consists of seven patterns where each one contains one *EReference* node and several nodes of type *EClass*. These patterns can be used to derive the following kinds of metrics: average, percentage, sum or total number of instances of a specific type in or for a selected instance. Please note that some patterns are designed to support inheritance in order to match child classes of classes, abstract classes or interfaces. For example, some patterns can derive the following kind of metrics: *number of instances of a child type in a selected instance*. Concrete examples are: *number of transition-place arcs in a selected Petri net*, *number of place-transition arcs in a selected Petri net*, and *total number of all arcs in a selected Petri net* (the sum of both metrics mentioned before).

Group "Two-edge patterns" The third group also consists of seven patterns. Here, each pattern contains two *EReference* nodes and several nodes of type *EClass*. These patterns can be used to derive the same kinds of metrics provided by group 2 as well as more complex ones. For example, a pattern is designed using two *EReference* nodes which have the same source *standard EClass* and the same target *standard EClass* node. The containment attribute of one *EReference* node is set to true whereas the containment attribute of the other one is set to false. This pattern can derive the following kind of metrics: *Number of "part"-instances in a selected "whole"-instance so that they have the same specific role specified by non-containment reference*. Some patterns of this group are designed to match child classes of different pattern nodes such as the child classes of the whole part node, the direct-part node or of both. The pattern in Figure 9 belongs to this group.

Further more-edge patterns can be defined in the similar way. We assume that our current patterns may be matched often over any given domain because their structures are vital and cardinally needed for representing several parts of any meta-model structure. Additionally, some metrics derived by more-edge patterns could be defined by composing metrics derived by less-edge patterns.

Group "Composed patterns" The last group consists of five patterns for deriving more complex metrics respectively more specific ones. It contains some patterns for deriving metrics used to calculate the sum (average, percentage) of the number of different kinds of instances having the same whole instance as for example *total number of transitions and places in a selected Petri net*. The patterns can also be used to derive metrics like *average number of places with respect to the number of transitions in a selected Petri net*. However, these kinds of metrics may not make sense for each domain.

So far, we defined 20 metrics patterns to derive 42 different kinds of metrics. However, we do not claim that this list is complete. Furthermore, the existing patterns could be used to produce further kinds of metrics. The 42 kinds of metrics are only some suggested ones. We can also think of combining several kinds of metrics. Additionally, we can also use the approach to design further metrics patterns by using, as an example, a different number of nodes and relations with different attributes values. Additionally, we can design patterns for producing metrics used to inquiry on attributes values of nodes.

5 Tooling

In this section, we present the Nas tool that we developed to automate the process of model basic metrics creation, i.e., metrics derivation and specification for any given domain, and to automatically generate a high-level tool specification as an Eclipse plugin thereafter. The entire tooling is based on the Eclipse Modeling Framework (EMF) [EMF, St08].

Nas Tool We developed an Eclipse-based tool, called *Nas Tool*, which uses metrics patterns to automatically find matches and to automatically derive, specify and generate basic metrics of any given domain modeled in Ecore thereafter. The metrics patterns are designed by Henshin [Ar10, Hen], a model transformation engine for EMF based on graph transformation concepts, as pattern-based rules. Each rule mimics the EMF abstract syntax of a structure to be matched in a given meta-model. During the pattern matching process, an isomorphic mapping from EMF node and edge symbols in the pattern to actual nodes and edges in the meta-model is computed. The Nas tool can be easily applied: The only input is a meta model in Ecore. The outcome of the Nas tool is a high-level tool specification as an Eclipse plug-in which comprises the following: A specification tool support for model metrics containing already a number of basic metrics specified by OCL being derived from the meta model. This tool support can be used to define further model metrics as compositions of existing ones. An application tool support for model metrics which can be used to calculate the defined metrics on concrete domain-specific models. The generated plug-in is used as input to EMF Refactor which is an Eclipse tool that supports metrics calculation, reporting and composition. Figure 10 depicts the use of the Nas tool in combination with EMF Refactor.

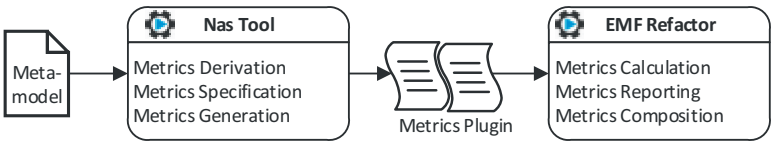

Fig. 10: The use of the Nas tool in combination with EMF Refactor

In addition, the Nas tool provides a view component for statistical information about derived metrics: It shows the number of pattern matches, the number of derived metrics for each applied pattern and the total number of matches and metrics.

Consequently, the Nas tool provides the following features:

- *Automation*: Creating (i.e. deriving, specifying and generating) basic model metrics automatically. Thus, the design and implementation time for each generated metrics is reduced.
- *Abstraction*: The Nas tool can be used to define and implement basic model metrics for any domain-specific language.
- *Simplicity:* The tool is easy to use, i.e., the user does have not to know about the metrics definitions, meta model structure and query languages.
- *Extendability:* The tool is extensible, i.e., it provides the ability to add further metrics patterns.

More information about the Nas tool, especially about its installation and the provided patterns, can be found at the accompanying web site of this paper [Nas].

6 Application Case

In this section, we demonstrate metrics generated by the Nas tool and compare them to the metrics provided by EMF Refactor [Ref] for a simple class modeling language (SCM) [AT13]. SCM represents the class diagram part of UML but it is much more simpler since it omits concepts like operations and association classes. Figure 11 shows the SCM meta model specified in Ecore.

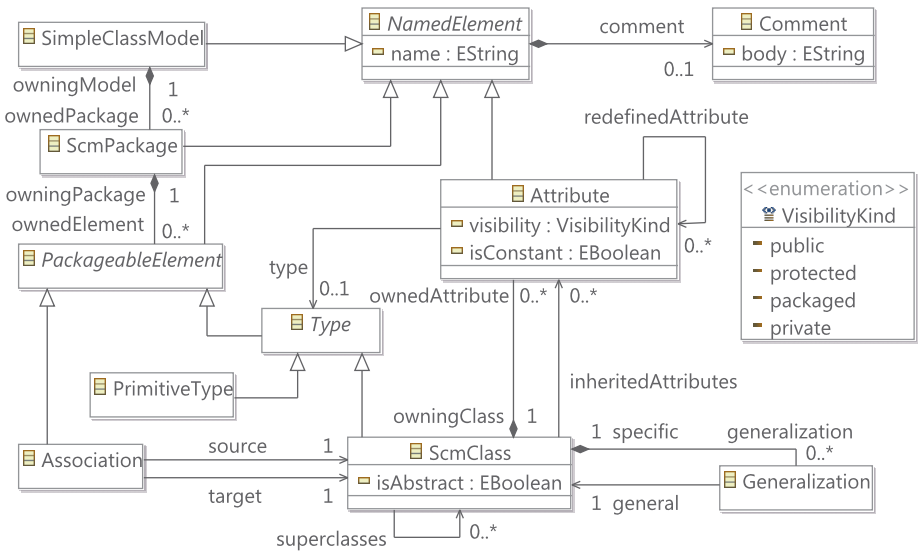

Fig. 11: The SCM meta model

The SCM metrics provided by EMF Refactor are manually specified and implemented using different perspectives and technologies like *Henshin*, *OCL* and *Java*. However, when

running the Nas tool on the SCM meta model, 127 basic metrics are automatically generated using only a few mouse clicks. 14 patterns are matched over the SCM model and some of them can generate several kinds of metrics. The 127 metrics are derived by altogether 72 matches of these metrics patterns. The number of pattern matches will be different from one domain to another one but we are convinced that the first six patterns (see [Nas]) have a high possibility to be matched over most given domains because the structure of each pattern of them could be considered as a simple basic structure.

After analyzing the SCM metrics provided by EMF Refactor, we found out that those metrics can be abstractly represented by 9 different metrics patterns. 6 of them are common with the patterns of the Nas tool whereas the other three patterns are too specific to the SCM model. The total number of the created metrics from these common patterns is 12 in EMF Refactor and 52 from the Nas tool.

Table 1 presents the SCM metrics provided by EMF Refactor and shows whether the metrics can be directly generated by the current version of the Nas tool, whether it can be defined by combining two generated metrics, or whether it cannot be generated. 12 out of 19 SCM metrics provided by EMF Refactor are also generated by the Nas tool. In addition, three metrics (i.e., 14, 15 and 17) are combinations of generated basic metrics. Hence, they profit from the Nas tool. We only need to combine two generated metrics using a mathematical operation, namely the division operation, like the metric called *Average number of attributes in concrete classes*.

In the following, we discuss the uncovered metrics: Metrics 2 and 18 could be derived by adding further metrics patterns, e.g., metric 2 can be derived by adding a pattern which takes incoming references into account. This metrics pattern shall be included into a future version of the Nas tool. Metrics 8 and 16 are too specific to the SCM model in the sense that they check a specific attributes value of the SCM model, e.g., if attribute *isAbstract* of *ScmClass* is set. In the future, we intend to consider ratio metrics for attribute values that are boolean or enumerations with literals.

The SCM metrics provided by EMF Refactor do not contain all the basic metrics of all SCM elements like the derived metrics from pattern 1, e.g., *number of all comments in the model*. The metrics in EMF Refactor are created using several different technologies and the process of creation took its time whereas creating the 127 SCM metrics generated by the Nas tool requires only a few mouse clicks. Please note that the non-covered metrics can be generated by adding new patterns or by extending some existing ones in the Nas tool. To summarize, the most of our selected patterns are matched with the SCM meta model and more than 78.5% of the metrics provided so far can be generated by the Nas tool or composed from generated ones.

Considering SDMetrics [SDM], a tool dedicated to the calculation of UML metrics, 96 metrics are provided by that tool, all specified by hand. We show that over 80% of them can be generated by the current version of the Nas tool or composed from generated ones. The rest of metrics could also be derived if the existence patterns were extended or new metrics patterns were added. More information about this comparison can be found at the web site of this paper [Nas].

Id	Metrics description	State
1	Number of direct children	covered
2	Number of external attributes with class type	not covered
3	Number of outgoing associations	covered
4	Number of incoming associations	covered
5	Number of associated classes	covered
6	Number of redefining attributes	covered
7	Total number of classes in the package	covered
8	Number of abstract classes in the package	not covered
9	Number of concrete classes in the package	covered
10	Number of associations in the package	covered
11	Total number of attributes in concrete classes	covered
12	Number of owned attributes in concrete classes	covered
13	Number of inherited attributes in concrete classes	covered
14	Average number of attributes in concrete classes	combination
15	Average number of associations per concrete class	combination
16	Ratio of the number of abstract classes	not covered
17	Ratio of the number of inherited attributes	combination
18	Number of equal attributes with other classes	not covered
19	Total number of model elements	covered

Tab. 1: The SCM metrics provided by EMF Refactor [AT13, Ref]

7 Related Work

To the best of our knowledge there is no directly related work on deriving model metrics from the corresponding meta model specification of a DSML. However, in this section, we discuss several topics being related to our work to some extent.

Model metrics. The problem of measuring the quality of models has been approached in several ways. Most of the presented metrics measure the quality of UML models. A survey of metrics applicable to UML models can be found in [GPC05]. Furthermore, in [La06, MP07], some general observations on managing quality, defining and reusing metrics for UML models are drawn.

The existing tool support for model metrics calculation is mainly aiming at UML and EMF modeling. Considering UML modeling, metrics calculation tools are integrated in standard UML CASE tools such as the IBM Rational Software Architect [RSA] and MagicDraw [MD]. A tool for UML metrics calculation is SDMetrics [SDM] (see above).

Related work in the context of quality metrics for meta models can be categorized into two groups: Firstly, work that deals with quality on the meta level, i.e., on meta models. Secondly, approaches that address quality on the instance level, i.e., on models. In [MA07], quality dimensions for MDD are derived. In [BV10], a taxonomy of meta model quality

attributes is presented. Both works use metrics to analyze the quality of the meta model instead of corresponding instance models. In [Vé06], a repository for meta models, models, and transformations is presented. The authors transfer metrics that were designed for class diagrams to meta models and apply them to contents of the repository. However, these works do not address the opposite direction to use structural information that is implicitly given in the meta model to derive basic metrics for instance models.

Metric definition and generation approaches. The closest relation to our work is presented in [YA14]. Here, the authors present a quality measurement framework for defining quality metrics at the meta model to measure the quality of conforming instance models. However, the motivation of this work is not to derive basic quality metrics like in our approach, but for evaluating a model by comparing it with a reference model which is motivated in the context of empirical studies, for example. In [AST10, Ar11, AT13] we present an EMF Metrics plug-in, as a part of EMF Refactor, supporting specification and calculation of model metrics for a given meta-model. Here, the derivation and specification of each model metric for each given meta-model have to be done manually by users. That work does not consider the automatic creation of model metrics for DSMLs.

In [Mo11], the authors present a model-driven measurement approach allowing modelers to dynamically add measurement capabilities to a DSML. The core of this work is to develop a metric specification meta model which enables to declare metric specifications as instance models. A metric specification model is taken as input to a prototype generator which outputs a full-fledge measurement software integrated into Eclipse. In this approach, the user should manually declare the metrics as instance models for each given meta model, whereas in our approach, the declaration of metrics will be automatically derived and specified from any given meta model.

In [Al09], the authors produce source-code representations of object-oriented applications. The generated representations should conform to a meta model that represents object-oriented languages such as Java and C++. A metric declarative language is developed to add new metrics without modifying the code of the framework. The metrics are executed on the generated representations. In that approach, the metric descriptions are declared concerning the object-oriented meta model, whereas in our approach, we derive metrics from any given meta model using characteristic patterns typed by Ecore (a meta-meta model) and using metrics descriptions.

8 Conclusion

In this paper we presented an approach for deriving basic metrics from the meta-model of a given domain-specific modeling language. In our work, 20 patterns are designed and described to derive metrics from any domain (meta-model) based on Ecore. The patterns are typed by the meta-meta model of EMF. Each pattern can be used to produce one or several kinds of basic model metrics.

We developed the Nas tool which takes the meta-model of any domain-specific language to automatically derive, specify and generate basic metrics based on the Eclipse technol-

ogy [Nas]. The use of the Nas tool is quite simple, only a few mouse clicks are required to create the metrics. Furthermore, the tooling provides an extension mechanism to add new custom metrics patterns. In this context, future work will concentrate on extending the existing metrics pattern base, supporting other metric kinds and more facilities for default calculation and compositions of metrics. Furthermore, we intend to figure out a vital set of metrics to be generated by analyzing, for example, user requirement specifications for the modeling language, to derive thresholds for the derived metrics, and to conduct further case studies.

References

[Al09] Alikacem, E-H; Sahraoui, Houari et al.: A Metric Extraction Framework Based on a High-Level Description Language. In: Source Code Analysis and Manipulation, 2009. SCAM'09. Ninth IEEE International Working Conference on. IEEE, pp. 159–167, 2009.

[Ar10] Arendt, Thorsten; Biermann, Enrico; Jurack, Stefan; Krause, Christian; Taentzer, Gabriele: Henshin: Advanced Concepts and Tools for In-Place EMF Model Transformations. In: Model Driven Engineering Languages and Systems, pp. 121–135. Springer, 2010.

[Ar11] Arendt, Thorsten; Kranz, Sieglinde; Mantz, Florian; Regnat, Nikolaus; Taentzer, Gabriele: Towards Syntactical Model Quality Assurance in Industrial Software Development: Process Definition and Tool Support. Software Engineering, 183:63–74, 2011.

[Ar14] Arendt, Thorsten: Quality Assurance of Software Models-A Structured Quality Assurance Process Supported by a Flexible Tool Environment in the Eclipse Modeling Project. PhD thesis, Philipps-Universität Marburg, 2014.

[AST10] Arendt, Thorsten; Stepien, Pawel; Taentzer, Gabriele: EMF Metrics: Specification and Calculation of Model Metrics within the Eclipse Modeling Framework. In: of the BENEVOL workshop. 2010.

[AT13] Arendt, Thorsten; Taentzer, Gabriele: A tool environment for quality assurance based on the Eclipse Modeling Framework. Automated Software Engineering, 20(2):141–184, 2013.

[BCW12] Brambilla, Marco; Cabot, Jordi; Wimmer, Manuel: Model-Driven Software Engineering in Practice. Synthesis Lectures on Software Engineering, 1(1):1–182, 2012.

[BV10] Bertoa, Manuel; Vallecillo, Antonio: Quality Attributes for Software Metamodels. Málaga, Spain, 2010.

[Ch95] Chamillard, Albert Timothy: An Exploratory Study of Program Metrics as Predictors of Reachability Analysis Performance. Springer, 1995.

[EMF] Eclipse Modeling Framework (EMF). http://www.eclipse.org/emf.

[GPC05] Genero, Marcela; Piattini, Mario; Calero, Coral: A Survey of Metrics for UML Class Diagrams. Journal of Object Technology, 4(9):59–92, 2005.

[Hen] Henshin. http://www.eclipse.org/henshin.

[HW14] Herrmannsdörfer, Markus; Wachsmuth, Guido: Coupled Evolution of Software Metamodels and Models. In: Evolving Software Systems, pp. 33–63. Springer, 2014.

[La06] Lange, Christian FJ: Improving the Quality of UML Models in Practice. In: Proceedings of the 28th international conference on Software engineering. ACM, pp. 993–996, 2006.

[MA07] Mohagheghi, Parastoo; Aagedal, Jan: Evaluating Quality in Model-Driven Engineering. In: Modeling in Software Engineering, 2007. MISE'07: ICSE Workshop 2007. International Workshop on. IEEE, pp. 6–6, 2007.

[Mc76] McCabe, Thomas J: A Complexity Measure. Software Engineering, IEEE Transactions on, (4):308–320, 1976.

[Mc82] McCabe, Thomas J: Structured Testing: A Software Testing Methodology Using the Cyclomatic Complexity Metric. US Department of Commerce, National Bureau of Standards, 1982.

[MD] MagicDraw. http://www.nomagic.com/products/magicdraw.html.

[MDN09] Mohagheghi, Parastoo; Dehlen, Vegard; Neple, Tor: Definitions and approaches to model quality in model-based software development - A review of literature. Information and Software Technology, 51(12):1646–1669, 2009.

[Mo11] Monperrus, Martin; Jézéquel, Jean-Marc; Baudry, Benoit; Champeau, Joël; Hoeltzener, Brigitte: Model-driven generative development of measurement software. Software & Systems Modeling, 10(4):537–552, 2011.

[MOF] Meta Object Facility (MOF). http://www.omg.org/mof.

[MP07] McQuillan, Jacqueline A; Power, James F: On the Application of Software Metrics to UML Models. In: Models in Software Engineering, pp. 217–226. Springer, 2007.

[Nas] Nas Tool. http://www.uni-marburg.de/fb12/swt/nassarn/nastool.html.

[Ref] EMF Refactor. http://www.eclipse.org/emf-refactor.

[RSA] Rational Software Architect (RSA). http://www-03.ibm.com/software/products/us/en/ratisoftarch.

[SDM] SDMetrics. http://www.sdmetrics.com.

[SJM92] Soo, Lee Gang; Jung-Mo, Yoon: An empirical study on complexity metrics of Petri nets. Microelectronics Reliability, 32(3):323–329, 1992.

[St08] Steinberg, Dave; Budinsky, Frank; Merks, Ed; Paternostro, Marcelo: EMF: Eclipse Modeling Framework. Pearson Education, 2008.

[SVC06] Stahl, Thomas; Voelter, Markus; Czarnecki, Krzysztof: Model-Driven Software Development: Technology, Engineering, Management. John Wiley & Sons, 2006.

[UML] Unified Modeling Language (UML). http://uml.org/.

[Va02] Van Solingen, Rini; Basili, Vic; Caldiera, Gianluigi; Rombach, H Dieter: Goal Question Metric (GQM) Approach. Encyclopedia of Software Engineering, 2002.

[Vé06] Vépa, Eric; Bézivin, Jean; Brunelière, Hugo; Jouault, Frédéric: Measuring Model Repositories. In: Proceedings of the 1st Workshop on Model Size Metrics (MSM'06) co-located with MoDELS. volume 2006, 2006.

[YA14] Yue, Tao; Ali, Shaukat: A MOF-Based Framework for Defining Metrics to Measure the Quality of Models. In: Modelling Foundations and Applications, pp. 213–229. Springer, 2014.