Measuring Models

Martin Monperrus, Jean-Marc Jézéquel, Joël Champeau, Brigitte Hoeltzener

Model-Driven Software Development: Integrating Quality Assurance (J. Rech, C. Bunse, eds.)

Martin Monperrus · Jean-Marc Jézéquel · Joël Champeau · Brigitte Hoeltzener

Measuring models

Abstract Model-Driven Engineering (MDE) is an approach to software development that uses models as primary artifacts, from which code, documentation and tests are derived. One way of assessing quality assurance in a given domain is to define domain metrics. We show that some of these metrics are supported by models. As text documents, models can be considered from a syntactic point of view i.e., thought of as graphs. We can readily apply graph-based metrics to them, such as the number of nodes, the number of edges or the fan-in/fan-out distributions. However, these metrics cannot leverage the semantic structuring enforced by each specific metamodel to give domain specific information. Contrary to graph-based metrics, more specific metrics do exist for given domains (such as LOC for programs), but they lack genericity. Our contribution is to propose one metric, called σ, that is generic over metamodels and allows the easy specification of an open-ended wide range of model metrics.

Keywords

Model-Driven Engineering, Metrics, Measurement, Genericity, Domain-Specific Metrics

1 Introduction

Model-Driven Engineering (MDE) is an approach to software development that uses models as primary artifacts, from which code, documentation and tests are derived. In this context, a model can be seen as the abstraction of an aspect of reality for handling a given concern in a specific domain. In MDE, the meaning of a model is itself defined with another model, called a metamodel. Complex systems typically give rise to more than one model because many aspects are to be handled.

One way of assessing quality assurance in a given domain is to define domain metrics from expert know-how, best practices or statistical analysis. As text documents, models can be considered from a syntactic point of view i.e., thought of as graphs. We can readily apply graph-based metrics to them, such as the number of nodes, the number of edges or the fan-in/fan-out distributions (see for example [1, 2]). However, these metrics cannot leverage the semantic structuring enforced by each specific metamodel to give domain specific information.

Contrary to graph-based metrics, more specific metrics do exist for given domains (such as LOC for programs), but they lack genericity. The lines of codes per method/function has been proven to be a software quality attribute [3]. As LOC or other code-centric software metrics, each domain has its own quality metrics.

An applied research program named *Measurement of Complexity* [4] lists more than one hundred *metrics of complexity* of importance in engineering. As an application of this program, a human review of textual documents has been done on four real world systems to compute the *metrics of complexity.* Complexity is not an issue of our investigation. However this program concludes on the need to facilitate the definition and computation of metrics and motivates this work.

The scope of our research is the definition of metrics at a higher level of abstraction than code, independently of the domain, while remaining rich enough for the domain expert. Our contribution is to propose one metric, called σ, that is generic over metamodels and allows the easy specification of an open-ended wide range of model metrics

The remainder of this chapter is organized as follows. We first give an introduction on model measurement in section 2. In section 3, we then introduce a generic metric, grounded in set theory and first-order logic. This metric is questionned with established metric property frameworks. We then discuss implementation issues. To show the genericity of our approach, we present in section 4 three case studies from various domains, and we show in section 5 that this also applies to the metamodel measurement. We finally discuss future research directions (section 6) and conclude.

2 State of the art

2.1 Dedicated model measurement

Metamodel measurement Ma et al. [5] compare different versions of the UML metamodel using OO metrics defined in [6]. Ma et al. [7] define patterns linked to the lifecycle of metaclasses, and study them on different versions of the UML metamodel. This work is similar in spirit to those made at the OO level [8–10].

MDE processes measurement Berenbach et al. [11] list a number of metrics for model driven requirements development and enounce some good practices. The Modelware project delivered three documents [12–14] in which several metrics about MDE processes are defined.

UML models measurement Previous works about the measurement of UML models follows the same decomposition as the UML artifacts themselves. Some authors address the measurement of class diagrams (see [15] for a survey), others the measurement of dynamic models [16, 17], component models (e.g., [18]), and OCL expressions [19,20].

Synthesis These works are dedicated to particular MDE artifacts i.e., metamodels, processes, UML models. They do not note that all this artifacts are models too, w.r.t. a metametamodel, a process metamodel or the UML metamodel. These contributions do not leverage this idea for defining a generic metric usable at any moment of product life-cycle, from requirements to implementation.

2.2 Metamodel based measurement of OO programs

Misic et al. [21] express a generic object-oriented metamodel using Z. With this metamodel, they express a function point metric using the number of instances of a given type. They also express a previously introduced metric called the *system meter*. Reissing et al. [22] extends the UML metamodel to provide a basis for metrics expression and then use this model to specify known metric suites with set theory and first order logic.

Harmer et al. [23] expose a concrete design to compute metrics on source code. The authors create a relational database for storing source code. This database is fed using a modified compiler. Metrics are expressed in SQL for simple ones, and with a mix of Java and SQL for the others. El Wakil et al [24] use XQuery to express metrics. Metrics are then computed on XMI files. Baroni et al. propose in [25] to use OCL to specify metrics. They use their own metamodel exposed in a previous paper. Likewise, in [26], the authors use Java bytecode to instantiate the Dagstuhl metamodel and specify known cohesion metrics in OCL.

These works are centered around the issue of OO metrics and are not metamodel independent. They show that it is useful to ground metrics into the semantic of source code i.e., its metamodel. It seems possible to generalize the idea and define precisely generic metrics on top of metamodels, set theory and logic. It is also to be noted that these approaches do not explore the modularity of metrics.

2.3 Generic metrics for OO measurement

Mens et al. [27] define a generic object-oriented metamodel and generic metrics. They then show that known software metrics are an application of these generic metrics. Alikacem et al. [28] propose a generic metamodel for object oriented code representation and a metric decription language.

These two contributions emphasize on a generic way to define metrics. However, they do not provide applications of the genericity outside the scope of OO metrics.

2.4 Generic model measurement

Saeki et al. [29] specify the definition of metrics in OCL as part of the metamodel. Saeki et al. do not attach the definition of metrics to any particular domain and underlines in a future research agenda the need for defining various domain metamodels, domain metrics and supporting tools.

Guerra et al. [30] propose to visually specify metrics for any Domain Specific Language and introduce a taxonomy of metrics. Tool support is provided in the Python and Atom3 environment. We share the motivation of this paper and provide further facts on the problem of model measurement and on the solution. Our case studies give a different perspective on the issue.

3 The generic σ metric

In this section, we present a model metric, called σ. The σ metric is a generic metric. It means that an executable metric is a specialization of σ. The genericity allows high level specification of metrics and a simplified implementation. Considering the Goal Question Metrics approach [31], the σ metric is a generic answer to a set of questions related to model quality:

Goal Improve the model quality from the modeler point of view.
Question #1 What model metrics can be related to functionality?
Question #2 What model metrics can be related to reliability ?
Question #3 What model metrics can be related to maintainability?

Metrics The family of σ metrics.

In this section, we first define a model in the model-driven engineering (MDE) sense in order to clearly ground the generic metric. Then, we introduce the notion of filtering function which is the kernel of the proposed generic model metric. We close the presentation with theoretical arguments and implementation issues.

3.1 Definition of a MDE model in set theory

A model, in the Model Driven Engineering (MDE) terminology, is at first glance a directed graph. A MDE model also contains information on nodes, sometimes refered as slots, which contain primary information. A model also embeds its structure i.e., the types of nodes, the types of edges, and the types of slots. The figure 1, inspired from [32], shows the different viewpoints on a model: a graph in the upper left part, a graph containing data in the upper right part, or a graph containing structured data in the lower left part where the structure is defined with a metamodel represented in the lower right part.

We define a model as:

Definition 1 *A model M is*

$$M = ((V; E; S); (C; R; A); (T_v; T_e; T_s))$$

where: V is the set of nodes, E a set of directed edges between nodes, which are elements of $(V \times V)$*, S is the set of slots, C is the set of classes, R is the set of relationships between classes i.e., elements of* $(C \times C)$*, A is the set of attributes of classes i.e., elements of* $(C \times \{boolean, numeric, etc.\})$*,* T_v *contains bindings between nodes and classes i.e., a set of elements of* $(V \times C)$*,* T_e *contains bindings between edges and relationships i.e., a set of elements of* $(E \times R)$*,* T_s *contains bindings between slots and attributes i.e., a set of elements of* $(S \times A)$*.*

Note that this definition includes support for languages such as Java or MOF, where one has classes and primitive types.

For convenience, we later use three functions:

- *source* which maps each edge to the source node of the edge;
- *target* which maps each edge to the target node of the edge;
- *type* which maps each node to a class $c \in C$.

3.2 The filtering functions

A filtering function is a function that tests a boolean condition on a given node. Applied to a set of nodes, it can be used as a filter.

Definition 2 *A filtering function* ϕ *is a morphism from the set of nodes to the truth values.*

$$\phi : V \to \{true; false\}$$

$$x \mapsto \phi(x)$$

ϕ is a boolean function, it thus can be a boolean formula of sub-filtering boolean functions. This function can be composed of an arbitrary unlimited number of conditions e.g., $\phi = \phi_1 \wedge (\phi_2 \vee \neg\phi_3)$. In figure 1, the filtering function $\phi(x) = (type(x) = City)$ is true for two out of three model elements: the *Frankfurt* node and the *Darmstadt* node.

We define the core filtering functions as: a test on the type, a test on a slot, a size test and a λ-test on a collection:

Test on the type This tests the type of an object of the model with respect to the name of a class. It is equivalent to the *isInstance* method of the Java class *Class*.
Test on a slot value This evaluates the value of a slot (a primitive type in Java, an EDataType in Eclipse Modeling Framework) with respect to a constant.
λ-Test on a collection This evaluates a sub-filtering functions on each member of the multiplicity element. This test is either a test *at least one* or *for all* and introduces a λ parameter.
Test on the size of a collection This evaluates the number of elements of a collection.

We chose these core filtering functions because they are sufficient for our case studies. Note that filtering functions are not closed in their definition and can include refinements so as to express more powerful statements. For example, it is conceivable to add regular expressions in slot tests on string values.

For instance, here are some examples of filtering functions refering to figure 1:

- $\phi(x) = (type(x) = City)$
- $\phi(x) = (x.name = "A5")$
- $\phi(x) = (size(x.roads) > 3)$
- $\phi(x) = (\exists\lambda \in x.destination | \lambda.name = "Darmstadt")$
- $\phi(x) = (\forall\lambda \in x.destination | \lambda.inhabitants > 100000)$

Filtering functions involve information from the metamodel e.g., *City* or *inhabitants*. They involve elements of C,R and A. Hence, they are at the same level as metamodels and are grounded into the structure of the models. The evaluation of a filtering function relies on the binding between a model and its structure i.e., T_v, T_e, T_s.

Set of Nodes As defined above, one of the components of a model is a set of nodes. It is possible to specify a subset of nodes X satisfying a filtering function. This is noted $SoN(\phi)(V)$ (SoN is the acronym for *set of nodes*).

Definition 3 $SoN(\phi)(V) = \{n \in V | \phi(n)\}$.

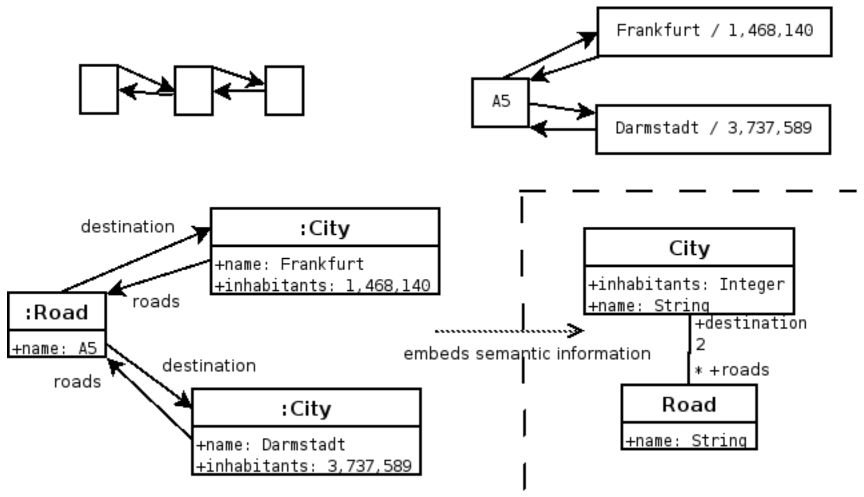

Fig. 1 Models from different points of view

3.3 Definition of the σ metric

The generic σ metric is derived from SoN and is the cardinality of a set of nodes given a filtering function. σ refers to the classical Σ mathematical symbol which denotes an iteration over a set of elements.

Definition 4 $\sigma_\phi(V) \in N = |SoN(\phi)(V)|$.

The σ metric characterizes an open-ended wide range of model metrics that are illustrated in sections below.

3.4 Theoretical validation

In this section, we confront the σ metric with theoretical matters: the type, the scale, the dimension and the reliability of the σ metric.

Type Several frameworks exists for validating software metrics e.g., [33–35]. We chose [33] to validate the metrics proposed in this paper because it is a formalized, yet convenient synthesis of previous works. Briand et al. enounced [33] formal properties for five types of software metrics: size, length, complexity, cohesion and coupling metrics. The σ metric satisfies the formal properties of size metrics *Size.1*, *Size.2* and *Size.3*:

Size.1 : Nonnegativity $\sigma(SoN(V)) \geq 0$ by definition of a set;
Size.2 : Null value if $SoN(V) = \emptyset \Rightarrow \sigma(SoN(V)) = 0$ by definition of a set;
Size.3 : Module Additivity $(V = V_1 \cup V_2$ and $V_1 \cap V_2 = \emptyset) \Rightarrow \sigma(SoN(V)) = \sigma(SoN(V_1)) + \sigma(SoN(V_2))$ (idem for E_1,E_2,E)

The σ metric can then be considered as a generic size metric applicable to any domain.

Scale The σ metric is a kind of count, hence is on an absolute scale. According to [36], all arithmetic analysis of the resulting count is meaningful, and according to [37], this scale permits a full range of descriptive statistics to be applied. This is true for a given filtering function, as discussed in the next paragraph.

Dimensional analysis Dimensional analysis aims to determine a consistent assignment of units. A dimension is a generalization of a unit of measure [37]. Since the σ metric is generic, it has no dimension associated with. It's a number of objects according to a filtering function. To this extent, a filtering function defines a dimension per se. Hence, one cannot directly perform arithmetics on different σ i.e., $\sigma_{\phi_1}(V)$ and $\sigma_{\phi_2}(V)$. For instance, summing $\sigma_{\phi_1}(V)$ and $\sigma_{\phi_2}(V)$ raises the same issue as summing a time and a length in physics. Derived measurement from σ metrics should be made carefully.

Measurement errors The σ metric does not have any measurement errors. It is theoretically reliable. Note that implementations still need to be tested or statically verified with the adequate methods to ensure its practical reliability.

Conclusion With respect to theoretical facts, the σ metric is a generic, reliable size metric on which descriptive statistics can be applied.

3.5 Implementation

We discuss below the reasons making the implementation of model metrics a difficult issue.

Non-programmer use The most appropriate person for defining domain metrics is a domain expert. He rarely has skills in programming. Hence, he needs to specify what he wants and to delegate the implementation to others . This dramatically increases the cost of definition and collection of model metrics. This observation is a strong motivation to define a simple and intuitive DSL, a coherent interface to be used by the domain expert for defining metrics.

Libraries / Framework Always for cost and productivity reasons, a good language for implementing metrics has the libraries to access models and their metamodels. It is very costly to develop an ad hoc and reliable parser, database connector, or binding to an existing modeling framework.

Ability to access to the metamodel The language or library for implementing metrics should include an easy way to navigate through the model and to access to the metamodel. For instance, considering a model element, there should be a way to access to the referenced elements as well as the metaclass and its attributes.

Non-intrusivity We have experienced that it is often tempting to pollute models with metrics concerns. For example, to add an attribute *marked : boolean* to the root class of the domain model, to mark visited objects. However, this practice violates the separation of concerns principle. A good metrics design practice is totally non-intrusive with respect to the semantic of the model. In the previous example, the need to mark visited objects implies the use of $Map : Object \rightarrow Boolean$.

Our implementation One can find in the literature several proposals for the implementation of model metrics (see section 2) e.g., Java, SQL, Python, Xquery, XML-based DSL, OCL, and a graphical language. Proposals mix some of these languages. These proposals do not adress all the issues cited above. We based our approach on the model-oriented programming language Kermeta [38].

Kermeta is a language based upon the EMF API. This facilitates the accesses to models and the navigation through models and metamodels. Furthermore, our industrial partners generally use Eclipse Modeling Framework (EMF) models. Kermeta features closures, which are of great help for the filtering functions. Indeed, a filtering function written in Kermeta is syntactically close to the underlying semantic hence very concise.

Polymorphism is useful to express modularized metrics. An example of code is shown on figure 2. An abstract class *SigmaMetric* encapsulates the generic code of the generic σ metric. A class *NumberOfCityMetric* is defined as a subclass of the SigmaMetric class. *NumberOfCityMetric* implements the filtering function ϕ and potentially delegates the definition of a subfiltering function

```
abstract class SigmaMetric {
  // generic implementation part of the sigma metric

  // specific part: an abstract method to be implemented
  operation phi(o : Object) : Boolean is abstract
} // end class

class NumberOfCityMetric inherits SigmaMetric {
  operation phi(o : Object) : Boolean is do
    result := (o.getMetaClass == City) and self.phi1(o)
  end

  operation phi1(o : Object) : Boolean is do
    return true
  end
} // end class

Class NumberOfCityConnectedToA5Metric inherits NumberOfCityMetric {
  operation phi1(o : Object) : Boolean is do
    return (o.asType(City).roads.exists{ x | x.name == "A5"})
  end
} // end class
```

Fig. 2 Implementation : excerpt of Kermeta code

ϕ_1 to subclasses by polymorphism. *NumberOfCityConnectedToA5Metric* overrides ϕ_1 to compute the number of cities connected to the road named *A5*.

Kermeta satisfies all the issues cited above, except the first one. Even if a filtering function written in Kermeta is syntactically close to its semantic, a domain expert can not feel comfortable with writing pieces of Kermeta code. It is outside of the scope of this contribution to specify visual or textual syntax usable by the domain expert.

The prototype involves two main abstract classes *ReflectiveWalk* and *SigmaMetric*. To define a σ model metric, the user just need to create a new class inheriting from *SigmaMetric* and to write the associated filtering function.

4 Applications of the σ metric

In this section, we present three case studies so as to illustrate the genericity and the feasibility of the σ metric. We first show that the σ metric allows the computation of logical lines of code in usual languages such as Java, the value-added of this approach is the ease of use compared to equivalent existing approaches. Then, we go beyond software metrics and consider non code centric artifacts such as requirements and system engineering metrics. The application of the σ metric in these various domains shows the genericity of the σ metric. Finally, since metamodels can be considered as models too, we present results of the σ metric at computing metamodel metrics in a full section.

4.1 Case study: lines of code (LOC)

Kan states that “the lines of code (LOC) metric is anything but simple” [39] (p.88). Indeed, there are numerous

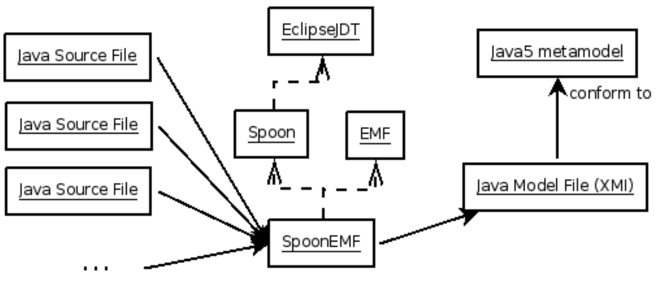

Fig. 3 The Java to EMF process

definitions of LOC, depending on authors and language (see [39]). Early LOC definitions follow a physical i.e., a representational point of view. An example is the count of the non-blank lines in source files. More sophisticated LOC definitions focus on logical statements (see [39,37]). This raises the technical issue of parsing the source files to access the semantics. In this section, we do not propose yet another LOC definition. We demonstrate that considering traditional programs as model, one gets essential LOC building blocks by applying the generic σ metric.

LOC semantic building-blocks are an application of the σ metric. They are numerous, among them are the number of methods, number of conditionals, number of blocks. For sake of readability, we do not list all of them. A representative part of them is shown in the results of our case study.

In this case study, we consider the Java programming language because of its wide diffusion. Furthermore, numerous open-source Java software packages are available. Since our implementation is based on the Eclipse Modeling Framework (EMF), we needed a way to transform Java source code into EMF models (as XMI files). We have used the SpoonEMF tool[1]. SpoonEMF is a binding between Spoon and EMF. Spoon [40] provides a complete and fine-grained Java metamodel where any program element (classes, methods, fields, statements, expressions, etc.) is accessible. This process finally transforms a whole Java software into a single XMI model file that can be natively processed with Kermeta.

The whole process is sketched on figure 3.

From this model, our prototype is able to compute the specific σ metrics values. For example, it produces the number of statements; number of assignments; number of conditionals; number of blocks.

To demonstrate the feasibility and the scalability of our approach, we have chosen five open-source Java software packages to be represented as models:

UmlGraph UmlGraph allows the declarative specification and drawing of UML class and sequence diagrams.

log4j log4j is a logging library. It provides an advanced service of logging, with emphasis on the performance of determining if a logging statement should be logged or not.

org.eclipse.osgi org.eclipse.osgi is the heart of the Eclipse IDE. It's the Eclipse implementation of the Open Services Gateway Initiative (OSGI) standards.

regexp regexp provides a regular expression Java library.

BCEL BCEL is a Byte Code Engineering Library (BCEL) intended to give users a convenient possibility to analyze, create, and manipulate (binary) Java class files.

For the sake of replication, the models used in this case study are available on the web[2].

In table 1, we show our results on the Java software packages listed above. Table 1 shows us that the σ metric is applicable to computer languages. Furthermore, it shows us that our approach scales with the model size (the org.eclipse.osgi model has 507798 model elements and 703360 references between them). Each line of the table compares the software packages under study with repect to a rigorously defined point of view. For instance, for a similar number of class, the BCEL software uses much less try/catch constructs than org.eclipse.osgi. Each column of the table is a vector that characterizes the software packages in a multi-dimensional space.

Measuring LOC with models as an application of the generic σ metric is easy. Spoon and SpoonEMF are components of our approach, yet have not been developed on purpose. One does not need to go inside a compiler or an interpreter, neither write a parser, which are both complex tasks.

One can object that much of the work has been moved on transformation from source code to models. But this task can be shared with other discplines such as aspect weaving or testing. Indeed, Spoon and SpoonEMF are not at all dedicated to metrics. Note that this burden does not exist for new languages when the designers make them directly available as models.

In this section, we showed that LOC building blocks are σ metrics. We presented our approach and our prototype to prove the feasibility of collecting LOC σ metrics on large-scale open-source software projects. Unlike traditional approaches, measuring LOC with models is easy. Collecting semantic metrics on source code could be done before. The added value of our approach is the simplicity. Firstly, it is easier to define and compute metrics on code represented as a model than on raw source files. Secondly, the application of a generic metric inside a framework to compute logical LOC is no more than a few lines of code i.e., the filtering functions.

[1] Please contact the INRIA Triskell team for further information

[2] `http://www.monperrus.net/martin/MDSDIQA-models.zip`

NumberOf	log4j-1.2.14	umlgraph-4.1	eclipse.osgi-3.2	regexp-1.4	bcel-5.2
TypeParameterReference	125	0	306	3	54
Method	1353	63	3389	119	2569
FieldReference	3713	373	12061	1156	7947
Catch	156	3	498	24	133
TypeReference	44874	5781	145149	6290	85155
For	104	8	867	32	379
LocalVariableReference	3379	388	16107	991	6503
Field	768	58	2183	137	983
Assignment	941	117	2785	253	1612
VariableAccess	5436	638	23221	1362	11415
Case	111	9	287	181	556
If	898	109	4392	263	1502
Parameter	1340	75	3621	159	2831
Constructor	295	11	391	22	545
ExecutableReference	7624	785	22913	1345	15409
Class	247	11	334	16	345
ParameterReference	2057	250	7114	371	4912
BinaryOperator	1874	269	8165	705	4922
ArrayAccess	127	63	1890	151	905
Return	932	60	3368	181	1416
Literal	3584	493	11747	1071	6691
Invocation	4842	484	13959	847	9392
Block	2979	164	7754	557	5283
LocalVariable	1327	126	5136	239	2117
ArrayTypeReference	1104	314	14461	724	5801
FieldAccess	3713	373	12061	1156	7947
NewClass	725	51	1840	127	1105
PackageReference	43691	5415	139303	6047	83183
UnaryOperator	264	33	1750	226	752
Try	139	4	530	19	129

Table 1 Results of the σ LOC metrics on large-scale Java software packages

4.2 Case study: requirements metrics

Since measurement is a tool to know if one is reaching the goal of building high quality requirements, several works have been done on defining requirements metrics (a survey can be found in [41,42]). The need for requirements metrics is also illustrated by the metric features of commercial tools e.g., Telelogic Doors. In this section, we aim to show that some requirements metrics are a special case of the generic σ metric.

Our methodology consists of taking a previous contribution on requirements metrics, extracting a metamodel, and identifying σ metrics.

Davis et al. define [43] a set of attributes that contributes to evaluate the quality of a requirements specification. 18 of the 24 quality attributes presented in this article have a mathematical metric formulation. These formulae are derived from the following metrics building blocks: the total number of requirements; the number of correct requirements; the number of stimulus input; the number of state input; the number of functions currently specified; the number of unique functions specified; the number of pages; the number of requirements that describe external behavior; the number of requirements that directly address architecture or algorithms of the solution.

All but the number of pages are concepts which are easily captured in a metamodel. In figure 4, we derive a metamodel which permits to compute the metrics above. This is a backbone for a bigger requirements metamodel. Except for the number of unique functions specified, all these metrics are σ metrics.

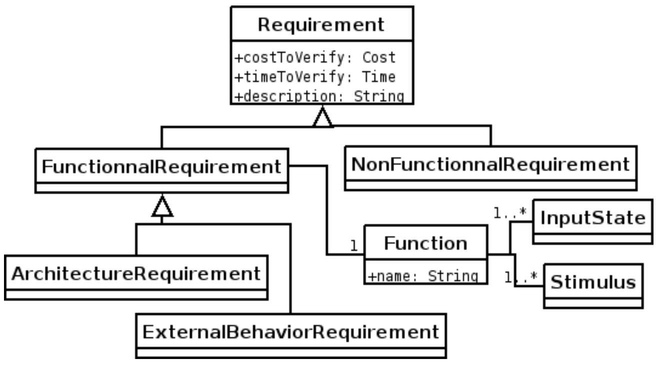

Fig. 4 The requirements metamodel extracted from [43]

Note that Davis et al. also use the following parameters: $C(R_i)$ is the cost necessary to verify presence of requirement R_i; $T(R_i)$ is the time necessary to verify presence of requirement R_i. The notations C and R (resp. *cost* and *requirement*) are from [43] and are totally different of the notations of section 3. The integration of this information in the metamodel is straightforward and represented in the figure 4.

It is possible, yet outside the scope of this chapter to apply this methodology to other requirements metrics contributions e.g., [44], so as to create a comprehensive requirements metamodel. We mainly aim to show the genericity of our contribution in the field of requirements engineering. A threat to our reasoning remains. One misses requirements models and case studies. Contrary to the previous section, where we have manipulated large scale models of source code , we are not able to produce metric values for this requirement metamodel due to the absence of models. However, the integration of model-driven requirements engineering in industrial processes shall solve this issue in a near future. Another solution to requirements metrics is to analyze natural language e.g., [45]. It is to be noted that these methods are primarily at a syntactic level i.e., deals with natural language. Our approach is totally at the model level and deals with the concepts, not the syntax.

4.3 Case study: system engineering metrics

As said in the introduction, an applied research program named *Measurement of Complexity* [4] driven by the DGA concludes on the need to automate the computation of system engineering metrics in a semantic manner for better reliability and affordable costs. This report grounds this case study.

This report lists 122 indicators that might measure a kind of complexity of the system. These indicators address a wide range of domains: requirements engineering; environment specification; software engineering; logical architecture; project management; mechanical and chemical architecture. This is a good artifact to illustrate the genericity of our contribution, genericity over the domain and over the product lifecycle.

For instance, from the metrics related to information systems, we have derived a metamodel i.e., we have listed the domain concepts of an information system e.g., Server, Protocol, Service, etc. The implementation in Kermeta is outlined in figure 5. This implementation of the metamodel associated with the σ metric enables the computation of most of the information system metrics identified in the report e.g., the number of protocols; the number of parallel databases; the number of file formats; the number of servers.

```
class InformationSystem inherits System
{ reference servers : Server
reference subSystems : InformationSystem[0..*] }
class Interface inherits Aspect::NamedElement,
        Aspect::VersionnedElement, Aspect::DescriptedElement {}
class NetworkService
{ reference protocols : Protocol[0..*] }
class PersistenceService inherits Service { }
class DataBase inherits PersistenceService
{ attribute replicated : DataType::boolean }
```

Fig. 5 Excerpt of the metamodel implemented in Kermeta

Exploring the whole list of indicators, it turns out than 45 out of 122 (37%) are metrics of the form *number of* i.e., an application of the σ metric. The table 2 shows that depending on the domain, the σ metric is more or less useful. The domain of software architecture is the best target, with 19 σ metrics out of 21 metrics. Thus, our approach really facilitates the definition and collection of metrics for model-based software

Requirements engineering Ex: Number of requirements without an associated test	12/28
Environment specification Ex: Number of variables	2/6
Software architecture Ex: Number of external protocols for interoperability	19/21
Logical architecture Ex: Number of configurations	9/34
Mechanical and chemical architecture Ex: Number of materials	1/21
Project management Ex: Number of stakeholders	2/12

Table 2 Complexity indicators which are an application of σ.

architectures. The domain of mechanical and chemical architecture is the worst target of the σ metric. To our knowledge, the reason is that these engineering fields are better described with mathematical models. Hence, the associated interesting metrics also are at a mathematical level. The domain of requirements engineering seems to be partly covered by the σ metric. However, most of the sixteen non-sigma metrics are totally subjective e.g., the distance of the required product from technological limits. In other words, computable requirements engineering metrics are well covered by the σ metric as discussed in the previous section.

This case study showed that a significant number of metrics identified outside the scope of model-driven engineering are applications of the σ metrics. Our contribution adresses totally 37% of metrics listed in the report. The other 63% metrics are mostly pure mathematical or physical ones, hence outside the scope of model-driven metric development. Our approach is promising since it solves the majority of logical metrics.

5 Genericity applied to metamodel metrics

The generic metric addresses the issue of defining and collecting metrics for a given domain. If the considered domain is metamodeling, the generic metric gives information on metamodels themselves. Our last case study is the metamodel measurement. Indeed, metamodels are important artifacts in Model Driven Engineering and we believe that it is essential to know their size, quality and complexity. For these purposes, metrics are to be defined, validated and implemented. This permits to give a numerical and objective vision of metamodels. Considering metamodels as models, we show that some of these metrics are an application of the σ metric presented above.

5.1 Metamodels: definition and implementation

There is no clear consensus on the definition of a metamodel. In [32], Kühne makes a contribution to the clarification of the definition, that *“may drastically simplify* *disputes about fundamental issues, such as the metamodel definition"*. We refer to this paper for the essence of a metamodel and follow a practical approach to define a metamodel: based on the EMOF specification [46], we outline the pratical differences between an EMOF metamodel and a class model, from typed object-oriented programming languages.

An EMOF model is composed of instances of classes from the EMOF package. The EMOF package is the result of the merge of five packages UML::Basic (from the UML infrastructure [47]), MOF::Common, MOF::Identifiers, MOF::Reflection and MOF::Extension. The main differences with object-oriented models are:

1. collections are explicitly typed;
2. collections have explicit lower and upper bounds;
3. collections are explicitly unique and/or ordered;
4. associations are defined as the binding between two references;
5. references can have containment role.

Note also that: EMOF differentiates references to primitive types (integer, boolean, etc.) and references to classes; EMOF allows multiple inheritance.

Ecore is part of the Eclipse Modeling Framework [48] (EMF) developped by IBM. It is an implementation of EMOF. The core EMF framework includes Ecore for describing models and runtime support for the models including change notification, persistence support with default XMI serialization, and a very efficient reflective API for manipulating EMF objects generically. The EMF framework includes generic reusable classes for building editors for EMF models and code generation facility capable of generating everything needed to build a complete editor for an EMF model.

Since Ecore is reflective and bootstrapped, Ecore metamodels can be considered as models. In this section, we leverage this facility to apply the generic model metric proposed above to the measurement of Ecore metamodels.

5.2 Direct application of the σ metric for metamodel measurement

Genero notes in [15] that unlike class measurement, the object-oriented *system metrics*, also called *package-scope metrics* [49, 50] have been little investigated. Since the notion of metamodel seems to be much more central in MDE than the notion of package in object-oriented programming languages, we aim to define metamodel metrics, not restricted to a given class i.e., metrics considering the metamodel as a whole. Thus, in this section, we define 6 global and simple metamodel metrics, taken or inspired from object-oriented *system metrics*. In the remainder of this section, we study the direct application of the σ metric for metamodel measurement i.e., metrics at the metamodel level so as to give a numerical and objective vision of metamodels. We consider:

NoC the number of classes.

NoD the number of primitive datatypes. A datatype is a primary information type e.g., byte, short, int, etc.

$TNoR$ the total number of references. It is the sum of the number of references of each class. In the EMOF terminology, a reference points to another class i.e., its type is a class and not a primitive type. The T (like the first letter of *total*) denotes that one considers the metamodel level (without the T, one considers the number of references per class), thus $TNoR = \Sigma_{i \in C} NoR_i$.

$TNoA$ the total number of attributes. It is the sum of the number of attributes of each classes. Note that we use the EMOF terminology: an attribute is a relationship between a class and a dataype, it defines a slot for primary information. The T denotes that one considers the metamodel level (without the T, one considers the number of attributes per class), thus $TNoA = \Sigma_{i \in C} NoA_i$.

$NoAC$ the number of abstract classes;

NoE the number of enumerations. An enumeration is a kind of datatype, hence $NoE \leq NoD$.

Since metamodels can be considered as models w.r.t. the metametamodel, let us consider Ecore metamodels as models in the Eclipse Modeling Framework (EMF). In this implementation, the metamodel metrics above are σ metrics. Here are the corresponding filtering functions (names refer to the ecore metamodel implemented in EMF):

NoC $\phi(x) = (type(x) = ecore :: EClass)$;
NoD $\phi(x) = (type(x) = ecore :: EDataType)$;
$TNoR$ $\phi(x) = (type(x) = ecore :: EReference)$;
$TNoA$ $\phi(x) = (type(x) = ecore :: EAttribute)$;
$NoAC$ $\phi(x) = (type(x) = ecore :: EClass$ and $x.abstract = true)$;
NoE $\phi(x) = (type(x) = ecore :: EEnumeration)$.

In the next section, we show that the generic σ metric can be a building block for metrics which take into account the specificities of metamodels w.r.t. object-oriented models.

5.3 The σ metric as a building block so as to take into account metamodel specificities

Existing object-oriented metrics do no take into account some modeling facilities available in EMOF metamodels. In this section, we use the *Goal Question Metric* approach [31] to define three new metrics that leverage these facilities so as to prove the ability of the σ metric to ground other metrics.

5.3.1 Goal

Improve the knowledge about EMOF metamodels to identify bad and good practices, patterns and templates. This identification will finally improve the quality of produced or refactored metamodels.

5.3.2 Questions

- How to characterize the use of associations in EMOF metamodels from the metamodel designer and user point of view ?
- How to numerically characterize the use of containment in EMOF metamodels from the metamodel designer and user point of view ?
- How to numerically characterize whether the EMOF metamodel is primitive data oriented or relationship oriented ?

The answers of these question can be the independent variables of an experiment where the dependent variable is a quality attribute (e.g., from [51]) of a MDE process. To this extent, our approach grounds the characterization of metamodel quality.

5.3.3 Metrics

The navigability metric The navigability metric involves the number of associations of the metamodel and $TNoR$ described above. The number of associations $TNoAss$ of the metamodel is a σ metric. The navigability metric is further named Nav.

Definition 5 $TNoAss = \sigma_\phi/2$ *where* $\phi(x) = (type(x) = ecore :: EReference$ *and* $x.opposite \neq null)$*;*

Definition 6 $Nav = (2 * TNoAss)/TNoR$

Properties $0 \leq Nav \leq 1$ since an association is made from two references i.e., $2 * TNoAss \leq TNoR$.

Interpretation If Nav = 0 the metamodel designer does not at all use EMOF associations and only uses simple references, if Nav = 1 the metamodel designer only uses EMOF associations i.e., all references are bound to the opposite one.

The containment metric The containment metric evaluates the use of containments in the metamodel. It is further named $Cont$. The containment metric involves four quantities A,B,C and D. A is the number of associations with the containment role. B is the number of associations $TNoAss$. C is the number of references with the containment role not part of an association. D is the number of references non part of an association.

- $\phi_A(x) = (type(x) = ecore :: EReference$ and $x.opposite \neq null$ and $x.containment = true)$;
- $\phi_B(x) = (type(x) = ecore :: EReference$ and $x.opposite \neq null)$;
- $\phi_C(x) = (type(x) = ecore :: EReference$ and $x.opposite = null$ and $x.containment = true)$;
- $\phi_D(x) = (type(x) = ecore :: EReference$ and $x.opposite = null)$;

Definition 7 $Cont = (A/B + C/D)/2$

Properties $0 \leq Cont \leq 1$. Proof: $0 \leq A, B, C, D$ since it is a σ metric. $A < B$ and $C < D$ since ϕ_A (resp. ϕ_C) is stronger than ϕ_B (resp. ϕ_D). Then $A/B, C/D \leq 1$. Finally, the division by 2 normalizes the metric to 1. If $B = 0$ (resp. $D = 0$), then $A = 0$ (resp. $C = 0$). Hence, if $B = 0$ (resp. $D = 0$), the whole term A/B (resp. C/D) is discarded.

Interpretation If Cont = 0 the metamodel designer does not use at all EMOF containment, if Cont = 1 the metamodel designer always uses containment i.e., all relationships have a container role.

Data quantity metric

Metric The data quantity metric is the ratio between the number of EMOF attributes (EAttribute) and the number of EMOF structural features (EStructuralFeature). It is further named Dat. It is based on global metamodel metrics defined above in section 5.2.

Definition 8 $Dat = TNoA/(TNoA + TNoR)$

Properties $0 < Dat < 1$ by definition.

Interpretation The data quantity metric is a kind of signature of the modeled domain and/or the modeling style.

Note that there is no a priori good or bad values for these 3 metrics. Future use and validation can clarify their meanings. In the next section, we present empirical results of the nine metrics presented.

5.4 Empirical results on real metamodels

5.4.1 Presentation of the metamodels

We aim to demonstrate the applicability of the generic metric for metamodel measurement i.e., how the generic metric fits to metamodel metrics. More and more metamodels are publicly available, as part of open source projects or standardization effort e.g., [47]. We have configured our generic prototype for the nine metamodel metrics previously exposed and computed them on the following metamodels:

AADL Architecture Analysis & Design Language metamodel, AADL is a standard for real-time embedded systems driven by the Society of Automotive Engineers - the corresponding Ecore implementation is part of the standard;

UML2 the Unified Modeling Language metamodel - the corresponding Ecore implementation comes from the UML2 project of Eclipse.

Ecore the EMF implementation of the EMOF metamodel;

XML Schema Definition XML Schema Definition metamodel - the Ecore implementation comes from the EMF tool which provides support for converting between Ecore and XML Schema models;

KDM Knowledge Discovery Metamodel of the OMG - the Ecore implementation comes from the Atlantic zoo[3];

Java the Java 5 metamodel - the Ecore implementation has been extracted from the Spoon tool [40] in our team.

For the sake of replication, the Ecore implementation of these metamodels used in this case study is available on the web[4].

5.4.2 Results

In table 3, we present the results obtained with our prototype.

It shows that the σ metric is adapted to the metamodel measurement; it also roots new metrics that leverage metamodel specificities; and it is implementable and scalable enough to be applied to real world metamodels.

The basic interpretation of this table is that NoC, TNoR, TNoA, NoDT enable an objective and numeric description of metamodels in a concise manner. That is to say, we can see without previous knowledge that all these metamodels are relatively different in their structure. NoAC and NoE are refinements of these metrics.

It also shows that the ratio between references and classes is discriminant: from more than two for XMLSchema to approximately one for KDM. We tend to think that there are two types of metamodels. The first type is element-dominant such as KDM, where the main goal is to explicit the concepts of a domain. The second type is reference dominant, such as XMLSchema, where the main goal is to describe relationships.

The metrics Nav, Cont and Dat are discriminant. We believe that these metrics give information on the modeled domain and the modeling styles and practices. However, the balance between the domain and the style cannot be determined. For instance, considering *Cont*, UML2 has a value of 0.50 while Java5 has a value of 0.01. To our opinion, this can be due to the fact that the Java5 modeler did not master the containment feature of Ecore, or to the fact that the Java5 programming language is not adapted to this modeling feature. The same reasoning concerns *Nav* and *Dat*.

We showed in section 3.4 that, from a theoretical point of view, the σ metric is closely related to size. This is confirmed in this domain of application, where intuitive metamodel size metrics e.g., number of classes, are σ metrics. Note that metrics where σ is used as a building block, such as *Cont* are not size metrics.

To conclude, we showed that the generic metric application on metamodel measurement is possible i.e., permits to define existing metrics and new metrics which leverage the metamodel specificities.

[3] `http://www.eclipse.org/gmt/am3/zoos/atlanticZoo/`
[4] `http://www.monperrus.net/martin/MDSDIQA-metamodels.zip`

6 Future Research Directions

Our contribution enables the easy definition of metrics and collection of metrics values. Thus, the main research direction is to leverage the generic σ metric to identify and empirically validate quality attributes in a given domain.

The study of metamodel metrics through the generic σ metric led to the definition of the metrics *Nav*, *Cont* and *Dat*. While our goal was to demonstrate the applicability of the generic metric, we also aimed at defining new and valuable metrics. This would be a better point in favor of generic metrics. However, we did not empirically validate the relationships between these metrics and software quality attributes. Future research is needed for the validation.

Finally, an issue is that the application of the generic σ metric is unaccessible to a non-programmer. This practice excludes the domain expert from defining and testing metrics in an autonomous manner. To this extent, a metric specific language is needed. As a perspective, a metric specific language will be studied so as to provide an intuitive notation and a user-friendly interface accessible to non-programmers. Defining and computing metrics should be totally transparent so as to unleash the domain analysis and creativity.

7 Conclusion and perspectives

In this chapter, we defined a generic metric that support the measurement of domain-specific attributes. The generic metric σ is defined using set theory and first order logic. It is the cardinality of a subset of model elements satisfying a filtering function. The theoretical validity of the generic σ metric is questionned in regard to type, scale, dimensional analysis and measurement errors. This shows that it is closely related to size.

To illustrate the genericity of the σ metric, we presented four case studies. We showed that the σ metric is able to precise the concept of source lines of code. By rising up Java source code at the model level, we were able to produce σ metric values on open-source projects including Eclipse-OSGI and Apache-Log4j. Two others applications encompass a wider scope than code-centric metrics. System engineering is engineering in the large. Numerous relevant metrics identified by system engineering experts are applications of the generic σ metric. This is an argument to go from non-semantic document centric system engineering to model driven engineering. In a similar manner, the generic σ metric permits to express metrics on requirements. The σ metric can also ground metamodel metrics, which are a solution to have a concise and objective summary of their internal complexity, size and quality.

Name	*NoC*	*NoAC*	*TNoR*	*TNoA*	*NoDT*	*NoE*	Nav	Cont	Dat
KDM	259	51	263	31	7	1	0.56	0.35	0.10
UML2	228	48	437	91	19	13	0.36	0.50	0.17
AADL	189	39	387	34	13	8	0.12	0.28	0.08
Java5	73	1	92	39	13	0	0.12	0.01	0.30
XMLSchema	57	22	125	98	28	20	0	0.16	0.44
Ecore	18	5	34	3	32	0	0.50	0.49	0.48

Table 3 Results on EMOF metamodels.

References

1. B. Edmonds, *Syntactic Measures of Complexity.* PhD thesis, University of Manchester, 1999.
2. J. Van Belle, "Towards a syntactic signature for domain models: Proposed descriptive metrics for visualizing the entity fan-out frequency distribution," in *Proceedings of SAICSIT 2002*, 2002.
3. B. A. Kitchenham, L. M. Pickard, and S. J. Linkman, "An evaluation of some design metrics," *Softw. Eng. J.*, vol. 5, no. 1, pp. 50–58, 1990.
4. P. Chretienne, A. Jean-Marie, G. Le Lann, J. Stefani, Atos Origin, and Dassault Aviation, "Programme d'Étude Amont Mesure de la compléxité (marché n°00-34-007)," tech. rep., DGA, 2004.
5. H. Ma, W. Shao, L.Zhang, Z.Ma, and Y.Jiang, "Applying OO metrics to assess UML meta-models," in *Proceedings of MODELS/UML'2004*, UML 2004, 2004.
6. J. Bansiya and C. Davis, "A hierarchical model for object-oriented design quality assessment," *IEEE Transactions on Software Engineering*, vol. 28, pp. 4–17, Jan. 2002.
7. H. Ma, Z. Ji, W. Shao, and L. Zhang, "Towards the uml evaluation using taxonomic patterns on meta-classes," in *Proceedings of the Fifth International Conference on Quality Software (QSIC'05)*, vol. 0, pp. 37–44, 2005.
8. M. Mattsson and J. Bosch, "Characterizing stability in evolving frameworks," in *TOOLS '99: Proceedings of the Technology of Object-Oriented Languages and Systems*, p. 118, IEEE Computer Society, 1999.
9. J. Bansiya, "Evaluating framework architecture structural stability," *ACM Comput. Surv.*, vol. 32, no. 1es, p. 18, 2000.
10. T. Gîrba, M. Lanza, and S. Ducasse, "Characterizing the evolution of class hierarchies," in *Proceedings of 9th European Conference on Software Maintenance and Reengineering (CSMR'05)*, pp. 2–11, IEEE Computer Society, 2005.
11. B. Berenbach and G. Borotto, "Metrics for model driven requirements development," in *Proceeding of the 28th International Conference on Software Engineering (ICSE '06)*, pp. 445–451, ACM Press, 2006.
12. Modelware Project, "D2.2 MDD Engineering Metrics Definition," tech. rep., Framework Programme Information Society Technologies, 2006.
13. Modelware Project, "D2.5 MDD Engineering Metrics Baseline," tech. rep., Framework Programme Information Society Technologies, 2006.
14. Modelware Project, "D2.7 MDD Business Metrics," tech. rep., Framework Programme Information Society Technologies, 2006.
15. 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.
16. M. Genero, D. Miranda, and M. Piattini, "Defining and validating metrics for uml statechart diagrams," in *Proceedings of QAOOSE'2002*, 2002.
17. A. L. Baroni, "Quantitative assessment of uml dynamic models," in *Proceedings of the doctoral symposium at the 10th European Software Engineering Conference held jointly with 13th International Symposium on Foundations of Software Engineering (ESEC-FSE'05)*, pp. 366–369, ACM Press, 2005.
18. S. Mahmood and R. Lai, "Measuring the complexity of a uml component specification," in *QSIC '05: Proceedings of the Fifth International Conference on Quality Software*, (Washington, DC, USA), pp. 150–160, IEEE Computer Society, 2005.
19. J. Cabot and E. Teniente, "A metric for measuring the complexity of ocl expressions," in *Model Size Metrics Workshop co-located with MODELS'06*, 2006.
20. L. Reynoso, M. Genero, and M. Piattini, "Measuring ocl expressions: a "tracing"-based approach," in *Proceedings of QAOOSE'2003*, 2003.
21. V. B. Misic and S. Moser, "From formal metamodels to metrics: An object-oriented approach," in *Proceedings of the Technology of Object-Oriented Languages and Systems Conference (TOOLS'97)*, p. 330, 1997.
22. R. Reissing, "Towards a model for object-oriented design measurement," in *ECOOP'01 Workshop QAOOSE*, 2001.
23. T. J. Harmer and F. G. Wilkie, "An extensible metrics extraction environment for object-oriented programming languages," in *Proceedings of the International Conference on Software Maintenance*, 2002.
24. M. El Wakil, A. El Bastawissi, M. Boshra, and A. Fahmy, "A novel approach to formalize and collect object-oriented design-metrics," in *Proceedings of the 9th International Conference on Empirical Assessment in Software Engineering*, 2005.
25. A. Baroni, S. Braz, and F. Abreu, "Using OCL to formalize object-oriented design metrics definitions," in *ECOOP'02 Workshop on Quantitative Approaches in OO Software Engineering*, 2002.
26. J. A. McQuillan and J. F. Power, "Experiences of using the dagstuhl middle metamodel for defining software metrics," in *Proceedings of the 4th International Conference on Principles and Practices of Programming in Java*, 2006.
27. T. Mens and M. Lanza, "A graph-based metamodel for object-oriented software metrics," *Electronic Notes in Theoretical Computer Science*, vol. 72, pp. 57–68, 2002.
28. E. Alikacem and H. Sahraoui, "Generic metric extraction framework," in *Proceedings of IWSM/MetriKon 2006*, 2006.
29. M. Saeki and H. Kaiya, "Model metrics and metrics of model transformation," in *Proc. of 1st Workshop on Quality in Modeling*, pp. 31–45, 2006.
30. E. Guerra, P. Diaz, and J. de Lara, "Visual specification of metrics for domain specific visual languages," in *Graph Transformation and Visual Modeling Techniques (GT-VMT 2006)*, 2006.
31. V. R. Basili, G. Caldiera, and H. D. Rombach, "The goal question metric approach," in *Encyclopedia of Software Engineering*, Wiley, 1994.
32. T. Kuehne, "Matters of (meta-) modeling," *Software and System Modeling*, vol. 5, no. 4, pp. 369–385, 2006.
33. L. C. Briand, S. Morasca, and V. R. Basili, "Property-based software engineering measurement," *Software Engineering*, vol. 22, no. 1, pp. 68–86, 1996.
34. A. C. Melton, A. L. Baker, J. M. Bieman, and D. M. Gustafson, "A mathematical perspective for software measures research," *Software Engineering Journal*, vol. 5, pp. 246–254, 1990.

35. N. F. Schneidewind, “Methodology for validating software metrics,” *IEEE Trans. Software Eng.*, vol. 18, no. 5, pp. 410–422, 1992.
36. N. E. Fenton, *Software Metrics: A Rigorous Approach.* Chapman and Hall, 1991.
37. B. Henderson-Sellers, *Object-Oriented Metrics, measures of complexity.* Prentice Hall, 1996.
38. P. A. Muller, F. Fleurey, and J. M. Jézéquel, “Weaving executability into object-oriented meta-languages,” in *Proceedings of MODELS/UML 2005*, 2005.
39. S. H. Kan, *Metrics and Models in Software Quality Engineering.* Reading, MA: Addison Wesley, 1995.
40. R. Pawlak, C. Noguera, and N. Petitprez, “Spoon: Program analysis and transformation in java,” Tech. Rep. 5901, INRIA, 2006.
41. M. Medina Mora and C. Denger, “Requirements metrics: an initial literature survey on measurement approaches for requirements specifications,” tech. rep., Fraunhofer IESE, 2003.
42. G. Kandula and V. K. Sathrasala, “Product and management metrics for requirements.,” Master’s thesis, Umea University, 2005.
43. A. Davis, S. Overmyer, K. Jordan, J. Caruso, F. Dandashi, A. Dinh, G. Kincaid, G. Ledeboer, P. Reynolds, P. Sitaram, A. Ta, and M. Theofanos, “Identifying and measuring quality in a software requirements specification,” in *Proceedings of the First International Software Metrics Symposium*, 1993.
44. R. J. Costello and D.-B. Liu, “Metrics for requirements engineering,” *J. Syst. Softw.*, vol. 29, pp. 39–63, Apr. 1995.
45. W. M. Wilson, L. H. Rosenberg, and L. E. Hyatt, “Automated quality analysis of natural language requirement specifications,” in *Proceeding of the PNSQC Conference*, 1996.
46. OMG, “MOF 2.0 specification,” tech. rep., Object Management Group, 2004.
47. OMG, “UML 2.0 superstructure,” tech. rep., Object Management Group, 2004.
48. F. Budinsky, D. Steinberg, E. Merks, R. Ellersick, and T. J. Grose, *Eclipse Modeling Framework.* Addison-Wesley, 2004.
49. M. Genero, M. Piattini, and C. Calero, “Early measures for UML class diagrams,” *L’Objet*, vol. 6, no. 4, pp. 489–505, 2000.
50. M. Xenos, D. Stavrinoudis, K. Zikouli, and D. Christodoulakis, “Object-oriented metrics - a survey,” in *Proceedings of the FESMA Conference (FESMA’2000)*, 2000.
51. ISO/IEC, *ISO/IEC 9126. Software engineering – Product quality.* ISO/IEC, 2001.