HAL
open science

Model-driven Generative Development of Measurement Software

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

To cite this version:

Martin Monperrus, Jean-Marc Jézéquel, Benoit Baudry, Joël Champeau, Brigitte Hoeltzener. Model-driven Generative Development of Measurement Software. Software and Systems Modeling, Springer Verlag, 2011, 10 (4), pp.537-552. 10.1007/s10270-010-0165-9 . inria-00504670

HAL Id: inria-00504670

https://hal.inria.fr/inria-00504670

Submitted on 21 Jul 2010

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

Model-driven Generative Development of Measurement Software

Martin Monperrus* Jean-Marc Jézéquel†,‡ Benoit Baudry‡
Joël Champeau§ Brigitte Hoeltzener§

Abstract

Metrics offer a practical approach to evaluate properties of domain-specific models. However, it is costly to develop and maintain measurement software for each domain specific modeling language. In this paper, we present a model-driven and generative approach to measuring models. The approach is completely domain-independent and operationalized through a prototype that synthesizes a measurement infrastructure for a domain specific modeling language. This model-driven measurement approach is model-driven from two viewpoints: 1) it measures models of a domain specific modeling language; 2) it uses models as unique and consistent metric specifications, w.r.t. a metric specification metamodel which captures all the necessary concepts for model-driven specifications of metrics. The benefit from applying the approach is evaluated by four case studies. They indicate that this approach significantly eases the measurement activities of model-driven development processes.

1 Introduction

Metrics offer a practical approach [25, 48] to evaluate non-functional properties of artifacts resulting from model-driven engineering (MDE) development processes. Ledeczi et al. showed [25], that a *use of the models is design-time diagnosability analysis to determine sensor coverage, size of ambiguity groups for various fault scenarios, timeliness of diagnosis results in the onboard system, and other relevant* ***domain-specific metrics***. More recently, a similar point of view is expressed by Schmidt et al. [48]: *in the context of enterprise distributed real-time and embedded (DRE) systems, our system execution modeling (SEM) tools help developers, systems engineers, and end users discover,* ***measure****, and rectify integration and performance problems early in the system's life cycle*.

Contrary to general-purpose programming language measurement software, domain-specific measurement software is not big enough a niche market; that is to say there are no measurement software vendors for specific domains. Hence, companies most often have to fully support the development cost of the model measurement software. Similarly to measurement software packages for classical object-oriented programs [28], model measurement software is complex and costly. Indeed, it must address the following requirements: the automation of measurement, the integration into a modeling tool, the need for extensibility and tailoring, etc. In all, the order of magnitude of the cost of model measurement software is several man-months [28, 50].

Our goal is to address the cost of measurement software for models by providing a generative approach that can synthesize a measurement environment for most kinds of models. In other words, we would like to have a prototype that allows the generation of measurement software for UML models, for AADL models, for requirements models, etc.

*Technische Universität Darmstadt
†University of Rennes
‡INRIA
§ENSIETA

Our proposal is to create a metric specification metamodel that captures important elements thta are required for the definition of metrics and that are independent of any particular domain. The definition of metrics for models consists in building metrics specifications that instantiate the metric specification metamodel. These specifications refer to elements of a domain metamodel and define metrics in a purely declarative form. We have developed a prototype that can analyze such specifications and generate an Eclipse plugin to compute metric values for an instance of the domain metamodel. The generated artifacts fully fulfill the requirements for measurement software presented above.

This whole approach is a model-driven measurement approach, called *MDM approach* in this paper. The approach offers a modeling language to declare metric specifications and add measurement capabilities to a DSML. This measurement approach is model-driven from two viewpoints: 1) it measures models involved in a model-driven development process; 2) it uses models as unique and consistent metric specifications, w.r.t the metric specification metamodel.

We evaluate the MDM approach with four application cases and a cross-domain analysis. The first application case is about the measurement of object-oriented software. It shows that the MDM approach is able to express existing software metrics and that it is productive from the viewpoint of a code size comparison. The second application case consists of measuring architectural models of embedded system. This application case uses the Architecture Analysis & Design Language (AADL) metamodel [47] and shows that a model-driven measurement approach is better than a document-centric approach from the viewpoint of measurement. The third application case is in the domain of model-based requirements measurement. It shows that the MDM approach enables the unification and the computation of heterogeneous metrics in a single framework. The last application case discusses measurement for systems engineering in a real industrial context. It shows that the MDM approach enables engineers to obtain valuable metric values about the system being built.

Existing work goes in the same direction [31, 15, 18, 17, 50]. Compared to them, our contributions are: 1) our metric metamodel encompasses a bigger variety of metrics 2) all elements related to the metric specification are metamodeled, including predicates on model elements 3) our evaluation is larger in breadth: we apply the same approach and use the same measurement software generator to four orthogonal domains.

The remainder of this paper is organized as follows. In section 2, we present the concept of model-driven measurement. Then, we present the metric specification metamodel in section 3. Application cases of the whole approach are presented in section 4. Section 5 discusses the quality of the approach. We finally discuss related works in section 6 and conclude.

2 How to Generate Measurement Software?

The measurement approach presented in this paper is intended to be applied to models of a model-driven engineering (MDE) software development i.e.; applied to models fully described by a metamodel, which is later called domain metamodel. Our intuition was [34] that it is possible to generate measurement software from an abstract and declarative specification of metrics. This abstract level for metrics can be defined by a metamodel as well. In other words, the measurement software is generated from a model of metric specifications.

Figure 1 presents our model-driven measurement approach as an UML activity diagram. The application of the MDM approach begins by considering a model space. This can be done in two ways. On the one hand, one can create a domain metamodel that exactly fits a given family of systems. On the other hand, one can choose an existing metamodel, for instance a standard one. Then, the MDM approach consists in specifying metrics as instances of a metric specification metamodel. This metric specification metamodel is the core of our contribution. It is presented in the next section. The metric metamodel is domain-independent i.e.; does not contain concepts specific to a domain. It only contains concepts related to metrics. The next step of the MDM approach is to identify the models that will be measured. It is possible to measure models created for other engineering activities (e.g.;

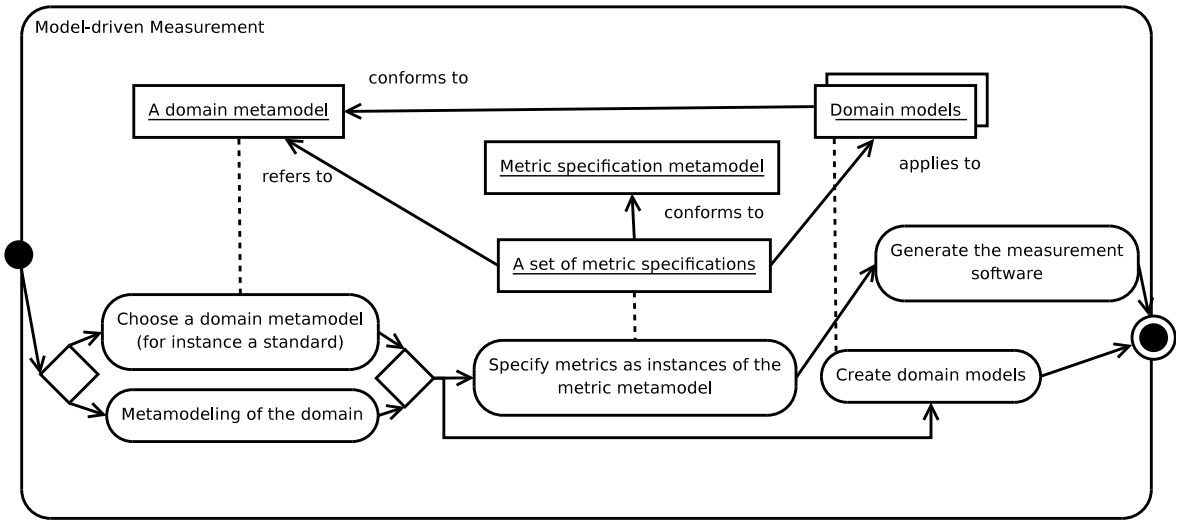

Figure 1: Model-driven measurement: actions and artifacts

simulation) or to create new domain models. Eventually, the MDM approach allows to really measure models by means of generative techniques described below.

Figure 1 also sums up the artifacts involved in the approach. The metric specification metamodel is the core artifact, it remains unmodified in whatever applications of the MDM approach. A set of metric specifications is an instance of the metric specification metamodel. A metric specification refers to concepts of the domain metamodel. Domain models are created conforming to the domain metamodel and can be measured thanks to the metric specifications[1].

The MDM approach is a solution to the issue of the measurement software cost because all the models involved ground code generation. The metric specification metamodel is used to generate the metric specification editor. For this activity, the MDM approach is based on existing techniques [51, 24, 6]. The metric specification model is used to generate the measurement software itself. This generation activity is fully automated. The generator is implemented as methods of the metric specification metamodel. These methods output the corresponding Java code. To sum up, a metric specification model is taken as input to a generator which outputs a full-fledge measurement software integrated into the generated modeling environment. Hence, the MDM approach gives an enhanced modeling environment; where the enhancement consists of the measurement features.

To conclude this overview, the prototype that supports the MDM approach has been developed in the Eclipse [2] and EMF [6] world. Metric specifications and domain models are EMF models w.r.t an EMF metamodel. The prototype generates an Eclipse plugin directly from the metric specifications. This plugin is fully-functional software integrated into Eclipse.

3 The Metric Specification Metamodel

The MDM approach consists in specifying metrics as an instance of a metric specification metamodel. The metamodel is a definition of the metric concepts. It grounds the complete generation of the measurement software from an instance of the metamodel, called a metric specification model. The concepts of the metric specification metamodel can be applied to any domain models. For instance, the same concept can be instantiated as a metric for implementation models, real-time models, software

[1]Note that domain models are not only created for measurement, but ground other engineering activities such as code generation, verification and validation, or model transformations.

[2]a generic development environment, see `www.eclipse.org`

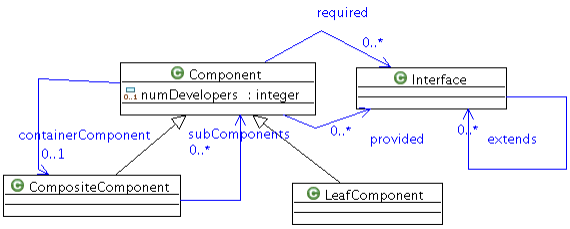

Figure 2: The domain metamodel used for demonstration purposes

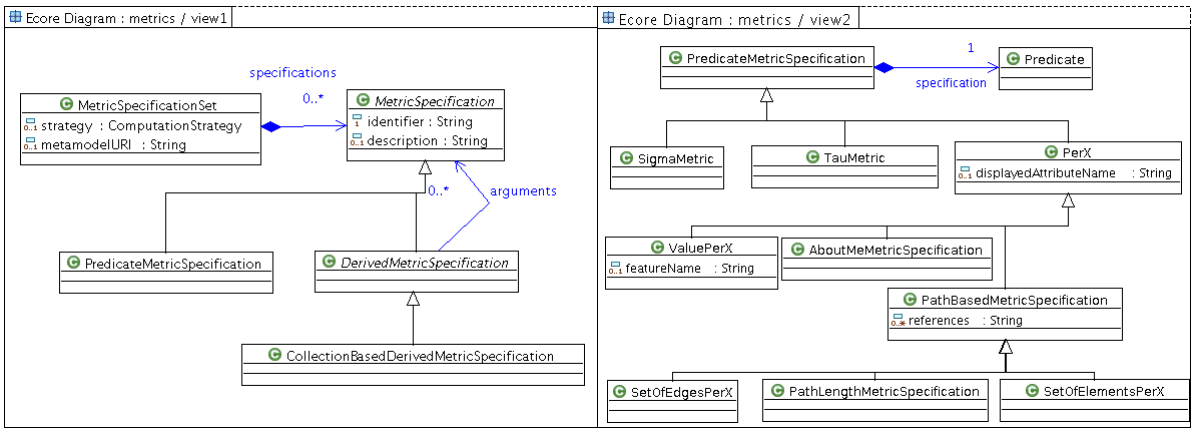

Figure 3: The core of the metric specification metamodel

architecture models or requirements models. In that sense, our metric specification metamodel is domain-independent [36]. It has been built following a bottom-up approach: from existing metrics to concepts of the metamodel. Hence, all concepts are instantiated several times.

Supporting example and syntax The presentation of the metric specification metamodel[3] will be supported by several examples. The corresponding examples consist of instances of the metamodel. Illustrative metrics are for a simple domain metamodel for software architecture depicted in figure 2. In this domain metamodel, the concepts manipulated are Component and Interface, and their relationships. This example metamodel is presented for demonstration purposes only.

A proto-textual syntax is used to textually represent the metrics, because a XMI-based model is not readable. Since our approach is model-driven, the most important artifacts remains the metamodel and its conforming models. That's why we use a *proto*-syntax and do not further define and discuss it. However, for sake of understandability, we tried to define an intuitive one, with explicit cognitive links between keywords and the metric specification metamodel.

Figure 3 presents the classes that structure the metric specification metamodel. A class *MetricSpecificationSet* is the root element. It contains several *MetricSpecification*. A *MetricSpecificationSet* represents a set of domain metrics collected by a domain expert. They can come from the literature, existing tools, company quality plans, expert know-how.

A *MetricSpecificationSet* refers to a domain metamodel (attribute *metamodelURI*), and to a *ComputationStrategy*. A computation strategy defines which parts of the domain model are measured. This can be a file, a predefined list of model elements or a transitive closure on all model elements. *ComputationStrategy* is placed at the level of the *MetricSpecificationSet*, and not at the level of *MetricSpecification*, because it defines the scope of measurement only, independently of the metric specifications.

[3]Note that the concepts of the metric specification metamodel will be further called classes.

A *MetricSpecification* is an abstract class that defines an atomic metric specification. Note that we use *MetricSpecification* and not Metric alone, to be able to differentiate a metric specification from a metric value. The modeling of measurement results i.e; metric values and their interpretation, is outside the scope of this metamodel. A *MetricSpecification* is sub-classed. The *DerivedMetricSpecification* class is used for metrics that are calculated in terms of other metrics, for instance arithmetic based metrics (e.g.; addition): the sub-classes of *DerivedMetricSpecification* are used to express complex metric formula. *CollectionBasedDerivedMetricSpecification* is the class that handles higher order metrics, mainly statistical operators, based on a set of metric values. Its subclasses are *Sum, Average, Median, Stddev* which are not represented in figure 3. The last subclass of *MetricSpecification* is *PredicateMetricSpecification*.

A *PredicateMetricSpecification* is an abstract class that contains a predicate. A predicate is a function from the set of model elements to the truth values. Predicate-related classes are presented later. Predicates are an important part of a metric specification since they precisely define the considered model elements for a given metric specification. We now consider the concrete classes that can be instantiated. The right part of figure 3 shows the classes that are instantiated as a metric specification for a given domain metamodel.

A *SigmaMetric* metric specification is the count of model elements that satisfy a predicate. The predicate can be as complex as needed. Similarly, the *TauMetric* is the count of model links; i.e. a link between two model elements. In this case, one has to specify the considered reference and if necessary, predicates for the link root and the link target. In the listing below, two instances of these classes considering the example domain metamodel of figure 2 are presented. A metric specification starts with the declaration of its type, a mandatory identifier, and an optional textual description. Then, there is the declarative part of the metric. For instance, a *SigmaMetric* declares a predicate and a *TauMetric* a reference name. Note that the predicate elements will be presented later.

```
metric SigmaMetric NOCompComp is
  description "too complex components"
  elements satisfy "this.isInstance(CompositeComponent)
          and this.required > 10
          and this.provided > 10
          and this.subComponents > 5"
endmetric

metric TauMetric NODeComp is
  description "The number of decomposition"
  link is "CompositeComponent:subComponent"
endmetric
```

The *PerX* class is an abstract class to specify metrics that are related to a given model element. For all selected model elements by a predicate, a value is returned. Hence, the measurement of a domain model thanks to a *PerX* metric specification does not return one unique value but several values (one per X, part of the selected model elements). The subclasses of *PerX* can be considered per se, or be given as input to a *CollectionBased* metric specification presented above, for instance the statistical operator average.

A *ValuePerX* metric specification represents metrics that can be obtained by considering only the model element satisfying a predicate. The *ValuePerX* class is abstract. It is sub-classed as *AttributeValuePerX*, *MultiplicityPerX*. An *AttributeValuePerX* is a metric whose value is directly given by an attribute of the model element. A *MultiplicityPerX* metric is directly given by the actual multiplicity of a reference. Let us now consider two instances of these classes.

```
metric AttributeValuePerX NODev is
  description "The number of developers per component"
  elements satisfy "this.isInstance(Component)"
  value is "this.numDevelopers"
endmetric

metric AverageMetric ANORI is
  description "The average number of required
              interfaces per component"
  input metric MultiplicityPerX is
    elements satisfy "this.isInstance(Component)"
```

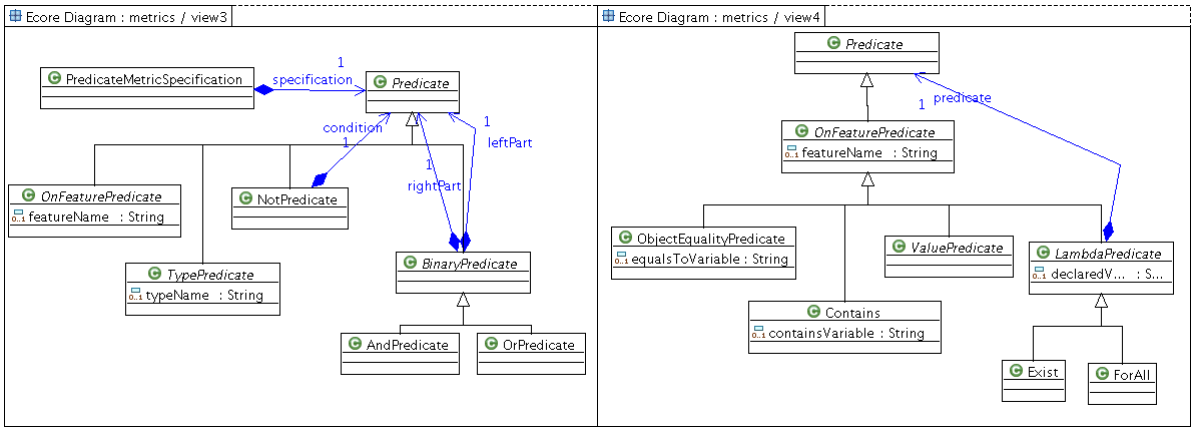

Figure 4: Predicates in the metric specification metamodel

```
    multiplicity of "this.required"
  endmetric
endmetric
```

A *SetOfElementsPerX* metric specification is the number of model elements obtained after performing a transitive closure. The *SetOfEdgesPerX* class is similar to the *SetOfElementsPerX* class, except that it counts the number of edges considered during the transitive closure.

A *AboutMeMetricSpecification* is a metric that considers a given object as primary variable. An *AboutMeMetricSpecification* contains internal metrics which can refer to a new variable *me*. When the metric is evaluated, for each selected element *AboutMeMetricSpecification*, internal metrics are evaluated by replacing the *me* by the selected element.

A *PathLengthMetricSpecification* is the maximum path length of a set of path starting from a root model element to other model elements. *PathLengthMetricSpecification* class is used to specify metrics that are based on a domain distance. Let us now consider instance examples of these metrics.

```
metric SetOfElementsPerX NOIT is
  description "The number of interfaces
             involved per component"
  -- we first select Components
  x satisfy "this.isInstance(Component)"
  -- we then compute the transitive closure
  -- of model elements linked with "required,provided,extends"
  references followed "required,provided,extends"
  -- and we count in this set the elements that are Interfaces
  elements satisfy "this.isInstance(Interface)"
endmetric

metric AboutMeMetricSpecification COUPLING is
  description "coupling to this interface"
  elements satisfy "this.isInstance(Interface)"
  internal metric spec is
   metric SigmaMetric COUPLING_ is
    description "used for COUPLING"
    elements satisfy "this.provided == __me__
                     or this.required == __me__ "
   endmetric
endmetric

metric PathLengthMetricSpecification DD is
  description "Depth in decomposition"
  x satisfy "this.isInstance(LeafComponent)"
  references followed "containerComponent"
endmetric
```

Metric specifications heavily rely on predicate. We now present the definition of predicates in the metamodel. This presentation is supported by figure 4.

A predicate can be composed of an arbitrary number of sub-predicates, in the same manner as a boolean function. Therefore, the metamodel contains the *AndPredicate*, *OrPredicate*, and *NotPredicate*. There are also classes that handle tests on the type of the model element. They are subclasses of the *TypePredicate* class. *IsInstance* tests the meta-class of a model element; *IsDirectInstance* tests the meta-class or one of its super-classes. The remaining class *OnFeaturePredicate* handles tests on features of a given model element. In this context, a feature means an attribute (e.g. a string "foo"), or a reference to another model element. *OnFeaturePredicate* class is further presented in the next paragraph.

The right part of figure 4 shows what the possible tests on a given feature are. *ObjectEqualityPredicate* enables us to test whether a reference points to a model element referred by a variable. *ValuePredicate* enables us to test the value of an attribute (e.g. a boolean equals to "false" or an integer equals to "13"). *LambdaPredicate* enables us to apply a predicate to all elements of a collection. It is sub-classed as *Exists* and *ForAll* to express first-order logic quantifiers. For sake of space, the following predicates do not appear on the figure. *Contains* enables us to test if a collection contains a model element referred by a variable. *MultiplicityPredicate* (not on the figure) enables us to test the actual size of a collection.

We have presented in this section the backbone of our model-driven measurement approach: the metric specification metamodel. An instance of this metamodel is a set of metric specifications formalized enough to generate the complete measurement software implementation.

4 Evaluation: Application Cases

In this section we present the application of the MDM approach into different contexts. For each application case, we present: 1) the motivation for measuring 2) the domain metamodel and 3) the metrics and the corresponding specifications w.r.t the metric specification metamodel. This section presents the facts that will be further analyzed in section 5.

4.1 Measurement of Java Programs

Motivation There is a large body of literature on software metrics. For instance, measuring software is helpful to predict defects [40], or to assess maintainability [8]. However, the design and implementation of measurement software is not an easy task [28]. If it is possible to consider classical software as models conforming to metamodels, the MDM approach could simply the measurement software.

Metamodel We have used the *SpoonEMF*[4] tool to transform a Java software to an *Eclipse/EMF* model. *SpoonEMF* is a binding between *Spoon* and EMF. Spoon [44] provides a complete and fine-grained Java metamodel where any program element (classes, methods, fields, statements, expressions, etc.) is accessible. *SpoonEMF* transforms a whole Java software package into a single XMI model file. The whole process is sketched on figure 5. To sum up, the domain metamodel here is the Java5 metamodel of *SpoonEMF*.

Metrics We have specified the well known Chidamber and Kemerer metrics [7] within the MDM approach. Hence, the metric specifications refer to the Java5 metamodel of SpoonEMF. Note that they are fully declarative.

Weighted Methods per Class: We consider the Basili et al's [2] version of WMC, which is the number of methods defined in each class.

```
metric MultiplicityPerX WMC is
    description "Weighted Methods per Class"
    elements satisfy "this.isInstance(CtClass)"
    reference is "Methods"
endmetric
```

[4] http://tinyurl.com/nxngpw

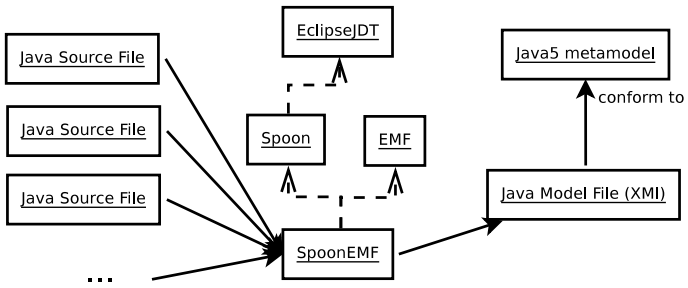

Figure 5: The Java to EMF process

Depth in Inheritance Tree per Class: DIT is defined as the maximum depth of the inheritance graph of a class.

```
metric PathLengthMetricSpecification DIT is
  description "Depth in Inheritance Tree per Class"
  x satisfy "this.isInstance(CtClass)"
  references followed "Superclass"
endmetric
```

Number of Children of a Class: NOC is the number of direct descendants for each class. It is a typical use of the *AboutMeMetricSpecification* metric type: there is no reference *children* in *CtClass*. *AboutMeMetricSpecification* fills this lack and enables us to correctly specify the metric.

```
metric AboutMeMetricSpecification NOC is
  description "Number of Children of a Class"
  x satisfy "this.isInstance(CtClass)"
  input metric SigmaMetric _NOC is
  elements satisfy "this.Superclass == __me__"
  endmetric
endmetric
```

Coupling Between Object Classes: CBO is a metric of coupling between classes of object oriented programs. It measures the number of references that point to a given class. To express CBO within MDM, we use *AboutMeMetricSpecification* to select each class and an internal *SetOfEdgesPerX* to count the number of pointers to each selected class (denoted as __me__).

```
metric AboutMeMetricSpecification CBO is
  description "Coupling Between Object Classes"
  elements satisfy "this.isInstance(CtClass)"
  input metric SetOfEdgesPerX _CBO is
  -- in SpoonEMF, all references (fields, method parameters, etc)
  -- are instances of CtTypeReference
  target satisfy "this.isInstance(CtTypeReference)
        and this.QualifiedName == __me__.SimpleName"
  endmetric
endmetric
```

Response For a Class: RFC is the number of methods that can be potentially executed in response to a message.

```
metric SetOfElementsPerX RFC is
  description "Response For a Class"
  elements satisfy "this.isInstance(CtClass)"
  x satisfy "this.isInstance(CtExecutableReference)"
  references followed "Methods,Body,Statements,
   Expression,AssertExpression,CaseExpression,
   ThenStatement,Condition,ElseStatement,Selector,
   Cases,Block,Finalizer,Catchers,AssertExpression,
   CaseExpression,Finalizer,Executable,
   DeclaringExecutable"
endmetric
```

Lack of Cohesion on Methods: It is not possible to specify the LCOM metric of Chidamber and Kemerer as an instance of the metric specification metamodel. The reason is that it involves as the core

metric component the concept of pair of functions. This concept is artificial w.r.t. the Java language and is not present in any form in the Java metamodel. However it still remains possible to manually develop the implementation of LCOM on top of the Java XMI model and the Java metamodel.

Conclusion The metric specification metamodel is rich enough to re-specify all but one Chidamber and Kemerer metrics with the MDM approach. In this context, the main artifacts are: a Java program considered as a model conforming to a Java metamodel, a set of metric specifications as a metric model instance of our metric specification metamodel. Section 5 will use this application case to discuss the productivity of the MDM approach.

4.2 Measurement of Embedded System Architectures

Motivation Our group has long-running collaborations with several companies delivering hardware and software for embedded systems. Most of them have development processes that include measurement. The measurement activities are mostly based on Excel sheets and informal systems description. Our partners are currently exploring model-based approaches to improve their development processes. In this context, it makes sense to replace the existing informal measurement activities by model-driven ones so as to get more precise measurement integrated into the modeling tools.

Metamodel For the application of the MDM approach at the software architecture level, we chose the Architecture Analysis & Design Language (AADL) metamodel as domain metamodel. AADL is a component based modeling language that supports early analysis of a system's architecture with respect to performance-critical properties through an extendable notation, a tool framework, and precisely defined semantics [13]. The main concepts of AADL are the concepts of system, device, bus, process, thread (see the AADL specification for more details [47]).

Metrics We had a pragmatic approach to choose the metrics to be applied on AADL models. We interviewed embedded system architects of 20+ years of experience in an automotive company. They listed the informal metrics they use. Then, we studied how to express these metrics w.r.t the AADL metamodel and as instances of the metric metamodel. The metric definitions are presented in the list below. D* stands for derived, P* stands for predicate based metric (cf. previous section).

P1 The number of devices (a computing resource in AADL);

P2 The number of processes;

P3 The number of threads;

P4 The number of non periodic threads;

P5 The number of orphan components i.e., that are not connected to at least one port (this applies to *SystemImpl, ProcessImpl*). Real-time systems architects really need to identify this problem;

D1 The mean number of processes per device is $P2/P1$;

D2 The mean number of threads per process is $P3/P2$;

D3 The schedulability difficulty indicator is $D1 * D2$. This is the number of tasks that the scheduler has to manage. Real-time systems architects recommend a strong risk warning to be raised if this metric is evaluated to more than 30;

D4 The load predictability is $P4/P3$. If all threads are periodic i.e., $D4 \rightarrow 0$ and if they are well defined in terms of CPU consumption, the load of the device is easily predictable. On the contrary, if they are mainly sporadic or aperiodic i.e., $P4/P3 \rightarrow 1$, it is extremely hard to predict the load. In this case, real time systems architects raise their security margins.

ID	Value	Interpretation
P1	5	saved in a database, used for cost and risk estimation afterwards.
P2	11	idem.
P3	69	idem.
P4	0	cf. D4
P5	0	the model is well-formed
D1	2.2	OK according to the empirical know-how.
D2	6.27	idem.
D3	13.8	is less than 30, acceptable.
D4	0	this indicates a low risk w.r.t. the load estimation.

Table 1: Metric values of an AADL model

Listing 1 shows an excerpt of the specification of these metrics using the MDM approach and metamodel.

Listing 1: Excerpt of metrics code

```
metric SigmaMetric P1 is
  description "Number of devices"
  x satisfy "this.isInstance(Component::DeviceImpl)"
endmetric

metric SigmaMetric P4 is
  description "Number of non periodic threads"
  x satisfy "this.isInstance(Component::ThreadImpl)
and this.propertyAssociation.exists{ x |
     x.propertyValue.select{z|~property::EnumValue.isInstance(z)}
     .exists{z |
        z.asType(~property::EnumValue).enumLiteral.name == "Periodic"}}
     or this.compType.extend.properties
     .propertyAssociation.exists{ x |
 x.propertyValue.select{z|~property::EnumValue.isInstance(z)}.exists{z |
          z.asType(~property::EnumValue).enumLiteral.name == "Periodic"}}"
endmetric

metric SigmaMetric P5 is
  description "Number of orphan components"
  x satisfy "this.isInstance(core::ComponentImpl)
and this.connections.eventConnection.size ==0
and this.connections.dataConnection.size ==0
and this.connections.eventDataConnection.size ==0
and this.connections.portGroupConnection.size ==0
and this.connections.dataAccessConnection.size ==0
and this.connections.parameterConnection.size ==0
and this.connections.busAccessConnection.size ==0"
endmetric

metric D4 is
  description "Load predictability"
  formula "P4/P3"
endmetric
```

Conclusion It is possible to use the MDM approach to measure AADL models. For instance, we have measured an open-source AADL model of a flight display system created by Rockwell Collins. Table 1 presents the obtained metric values. We will further discuss this table in section 5.

4.3 Measurement of Software Requirements Specifications

Motivation Measuring Software Requirements Specifications (SRS) allows to identify risks and flaws very early in the system life cycle [19]. For instance, the Capability Maturity Model (CMM) of the Software Engineering Institute emphasizes on the need for requirements measurements: *measurements*

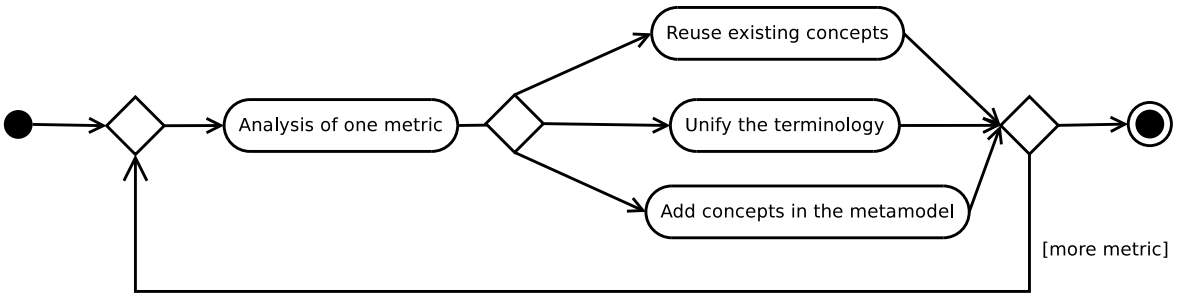

Figure 6: Metric-driven metamodeling of requirements.

are made and used to determine the status of the activities for managing the allocated requirements (*Key Practices of the Capability Maturity Model* [43, p. 98]). Also, according to Zave [54], requirements measurement is an important subfield of research on requirements engineering.

In this section, we show how the MDM approach is able to contribute to the state-of-the-art of requirements measurement. In a nutshell, we have collected the most influential papers of the literature related to requirements metrics [4, 10, 9, 27, 26, 22, 49, 23, 11, 5], we have defined a metamodel to structure the concepts that are involved in measuring requirements and we have created a consolidated list of requirements metrics from the selected papers. Finally, since all metrics are related to concepts of the requirements metamodel, we have specified the requirements metrics of the literature as instances of the MDM metamodel.

Metamodel Figure 6 shows the process we followed to create the requirements metamodel. It is a UML activity diagram. One by one, we analyzed each requirement metric in order to find out whether the concepts or relationships already exist in the requirements metamodel. If not, we added it to the requirements metamodel. This small number of classes shows that the majority of metrics deals with the same requirements concepts. In other terms, not all new analysis of a metric triggered the addition of a new concept. Somehow, the core of the requirements metamodel has been built only by analyzing the first half of metrics.

While some concepts were obviously similar, sometimes we had to unify some terms. It means that certain metrics of the literature deal with concepts that are similar yet named differently. Eventually, we obtained a metamodel that captures the common requirements modeling concepts and the relationships between these concepts. Again, every metamodel element has been created due to the need of the concept in a requirements metric definition. The application of this metric-driven metamodeling approach for requirements engineering ended up with a metamodel with 36 classes.

Metrics We have consolidated a list of 78 requirements metrics (see [33]) from the literature. Then, we have specified them using the MDM approach, i.e. expressing them as instances of the MDM metric metamodel. In the following, we elaborate on a sample of three of theses metrics formally specified with the MDM approach.

Total number of requirements: In MDM, a *SigmaMetric* counts the number of model elements satisfying a predicate. The total number of requirements is expressed as a *SigmaMetric*. Note that since our requirements metamodel handles the version history, we have to select only current requirements. If this.currentVersion is set, it means that the requirement has been overriden by a new one, hence we count only those which are not linked to a current version.

```
metric SigmaMetric 01_NOR is
  elements satisfy "(this.isInstance(Requirement)
     and this.currentVersion==void)"
  endmetric
```

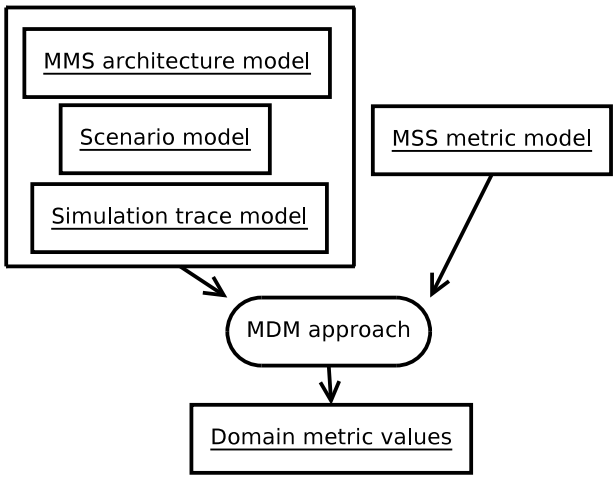

Figure 7: The MDM approach applied on different maritime surveillance system models

Number of Computer Software Configuration Item (CSCI) linked to a requirement: In MDM, a *SetOfElementsPerX* metric counts the number of elements linked to a root element by a path of a certain kind. Hence, this is specified using three predicates: 1) a predicate on the root element; 2) a predicate on the counted element, 3) a predicate on the path, which lists the references that can be followed. Note that if there are several elements matching as root element, we obtain one metric value per root element. The number of CSCI linked to a requirement counts the number of CSCI linked to a requirement by references of type *refinedIn* or *allocatedTo*.

```
metric SetOfElementsPerX 13_N is
  elements satisfy "this.isInstance(Requirement)"
  count "this.isInstance(CSCI)"
  references followed "refinedIn,allocatedTo"
endmetric
```

Degree of decomposition per requirement: In MDM, a *PathLengthMetricSpecification* gives the size of the longest path from a root element following a certain path. It is specified using two predicates: 1) on the root element and 2) on the path. The degree of decomposition per requirement can be specified as a *PathLength* metric w.r.t. the *moreAbstractDescription* reference between requirements (cf. metamodel).

```
metric PathLength 18a_N is
  elements satisfy "this.isInstance(Requirement)"
  references followed "moreAbstractDescription"
endmetric
```

Conclusion The application of the MDM approach to the problem of requirements metrics allows to generate software for measuring requirements. Model-driven measurement of requirements allows the unification and the computation of heterogeneous metrics in a single framework.

4.4 Measurement of Maritime Surveillance System Models

Motivation Thales Airborne Systems, is a world-wide company that designs and develops maritime surveillance systems. A maritime surveillance system (further called MSS) is a multi-mission system. It is intended to supervise the maritime traffic, to prevent pollution at sea, to control the fishing activities, to control borders. It is usually composed of an aircraft, a set of sensors, a crew and a large number of software artifacts. The number of functions, the relationships between hardware and

software components and the communication between the system and others entities (base, other MSS) indicate the complexity of the system.

The earliest phase of the development of maritime surveillance systems is called systems engineering. System engineering processes at Thales are largely document centric. They heavily use MSS simulators which are mainly code centric. The problem is that it is very costly to develop measurement software on top of either documents or code, in order to obtain operational metrics for MSS. By applying the MDM approach at the systems engineering level, our motivation is to enable engineers to obtain: 1) metric values that estimate operational performance of the system; 2) metric values that are used to dimension the system and its architecture.

Therefore, we tried to obtain MSS domain metrics with our approach. In two previous papers [35, 37], we presented a model-driven simulator for maritime surveillance systems. In this paper, we measure the models involved in this model-driven simulator thanks to the MDM approach.

Metamodel There are no standard metamodels for maritime surveillance systems. Consequently, we created a domain metamodel for maritime surveillance systems. The final metamodel is divided into three packages that address different concerns. More information on the metamodels can be found in [35, 37]. The MSS system architecture package contains architectural concepts. It is divided in five packages following a functional decomposition. The scenario package contains all the needed classes to represent a tactical situation. A model, instance of this metamodel, specifies the surveillance zone, the number and types of objects that are in the zone. For each object is specified a trajectory including the speed of the object. The operational event package contains classes that represent events that have a semantic with respect to a simulation at the system engineering level.

Metrics To choose the domain metrics, we studied the legacy simulators and discussed with systems engineers of the company. We agreed on 16 metrics to be implemented with the MDM approach. The domain metrics in the context of the MMS development at the system engineering are of different types. The types follow the same functional decomposition as the packages.

System architecture metrics (5 metrics) These metrics are related to the architecture itself e.g.; the number of sensors in the system. These metrics are mostly dedicated to cost estimation and planning.

Scenario metrics (4 metrics) These metrics are related to the tactical operation in which the system will evolve (for instance, the number of boats in the surveillance zone). The system dimensioning depends on such domain metrics.

Simulation trace metrics (7 metrics) These metrics are computed on simulation trace. They are estimations of the system properties; for instance the ratio of detected ships during the surveillance mission.

Each of these domain metrics has been specified as instance of the metric specification metamodel.

Conclusion We have applied the MDM approach in the context of maritime surveillance system development at the system engineering level. We have sketched the main components of the approach: the domain metamodels, models and metrics (please refer to [33, 37] for more details). As illustrated in figure 7, the MDM approach enables system engineers to obtain domain metric values for the daily activities, for instance: how many boats could the system detect? what is the estimated cost of the system based on its architecture? This comprehensive and consistent way of measuring system engineering models is an added value compared to costly ad hoc developments of measurement software which consider either document-centric or code-centric artifacts.

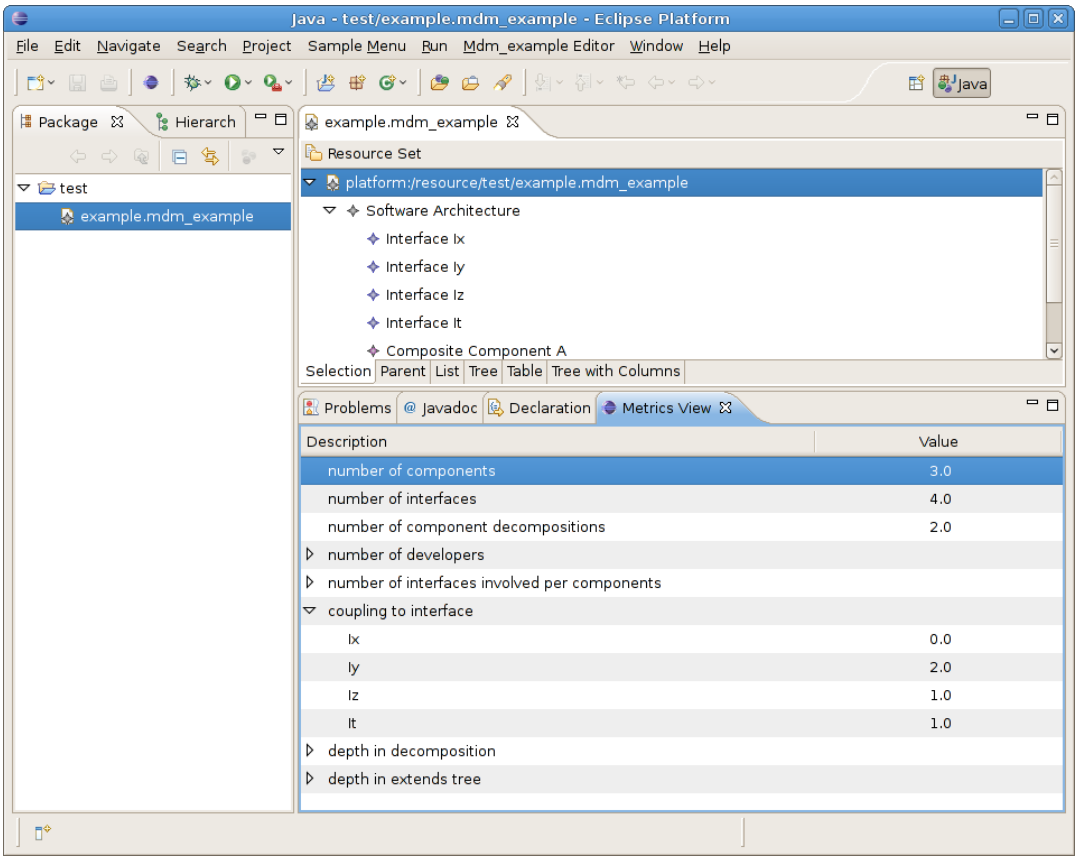

Figure 8: Generated and integrated measurement software

5 Evaluation: Characteristics

The MDM approach enables software and system engineers to measure any kind of models as long as they are structured by a MOF metamodel. The approach consists of generating full-fledge measurement software from an abstract specification of metrics. In this section, we present a cross-domain evaluation of the MDM approach, based on the application cases presented in section 4.

5.1 Is it a full generative approach?

The MDM approach is a *specify & generate* approach. The generated measurement software is a plugin that runs into the Eclipse development environment. The generated software is complete, runnable and does not have to be modified by hand (from generated Java code to Eclipse configuration file, e.g. Manifest and bundle XML files).

For dynamic metrics (i.e. those that depend on system execution), the code generator does not instrument existing systems. In such cases, the user has to provide instances of a trace metamodel to the generated measurement software (cf. 4.4).

Model	AADL Flight Sys.	Java eclipse.osgi-3.2
# model elements	30400	516413

Table 2: Biggest measured models

5.2 Is the generated measurement software user-friendly?

The generated measurement software is a full-fledged Eclipse plugin. It can be simply deployed in the Eclipse environment containing the model editor and other model-driven plugins (e.g. a model interpreter, a code generator, etc.) A screenshot of the generated measurement software is shown in figure 8. The models that are measured are instances of the domain metamodel of section 3 (a simple example software architecture metamodel). The Eclipse environment contains a model editor generated by EMF [6], at top right-hand side of the figure. The measurement software proposes two new features. First, a measurement interface as an Eclipse *view*, at the bottom right-hand side of the figure. Second, a right click on a domain model proposes a "Measure" action, which computes the values of the specified metrics. Hence, the metrics view (bottom right-hand side) shows the metric values of the metrics presented in section 3 for the model displayed in the model editor (top right-hand side).

5.3 How fast is the generated software?

The generated measurement software is able to measure models composed of thousand of model elements. Table 2 shows the size of different models that have been measured with the MDM approach. For instance, the AADL model of a flight system that we measured contains 30400 model elements.

Our experience in using the prototype implementation showed that the measurement software is able to measure these large models in a few seconds.

5.4 Does the MDM approach improve the measurement activities?

The MDM approach improves the measurement activities of the system development processes. Our application cases support these claims.

In the context of embedded system architectures (see section 4.2), using the MDM approach is better than ad hoc Excel measurement and reporting. The same AADL model grounds system analyses, process analyses and code generation. System analyses are achieved with several techniques including metric based criteria (e.g., in table 1, the schedulability difficulty indicator D3 is lower than the empirical bound 30). Process metrics of table 1 (P1, P2 and P3) can be automatically saved in a database in order to ground future cost and risk estimation models.

In the context of requirements measurement (see section 4.3), the model-driven design of both requirements and metrics is a solution to the problem of analyzing natural language. In this perspective, the MDM approach is a step towards the systematic measurement of requirements, as advocated by several authors (e.g. [43, 19, 54]).

In the context of systems engineering of maritime surveillance systems (see section 4.4), the MDM approach enables engineers to obtain system and simulation metrics that were very difficult to obtain before. Having explicit metamodels involved in the system simulations facilitates both the design of the simulator [37] and the measurement of system characteristics.

Overall, the MDM approach eases the production and the maintenance of measurement software.

Domain	# metrics
Java	5
AADL	9
Requirements	78
Maritime Surveillance Systems	16

Table 3: Metrics expressed with the MDM approach

5.5 Is it cheaper to apply the MDM approach or to manually develop measurement software?

The application cases of section 4 give evidences that the MDM approach decreases the cost of measurement software.

The specification of metrics for java programs is 24 lines of code with our textual declarative language. The generative step of the MDM approach generates a Java program of around 100 lines of code (LOC). This Java program uses a model measurement library of around 3000 LOC, that is not dedicated to Java but used in all the MDM approach instances. This library seats on top of *SpoonEMF/EMF/Eclipse*. By comparison, the *Metrics*[5] Java measurement tool is more than 20000 lines of code manually coded, while implementing similar metrics. Although a quantitative comparison is not possible, the difference in order of magnitude suggests that our approach reduces the cost of measurement software. Indeed a significant part of the accidental complexity of measurement software (e.g. UI integration) is encapsulated once in the measurement library and in the code generator of the prototype implementation of the MDM approach.

We implemented 16 domain metrics for maritime surveillance systems using the MDM approach within 1 week (one day for the metrics, 4 days to solve bugs in the measurement software generator prototype). For comparison, a previous project that developed a similar set of domain metrics for an agent-based simulator took several months. Interpreting these numbers in a qualitative manner indicates that having a conceptual and technical framework for expressing metrics do help their definition and implementation.

5.6 How generic is the metric metamodel?

To evaluate the genericity of the metric metamodel, we applied the approach in different contexts. In section 4, we have presented four application cases of the MDM approach: measuring Java models, maritime surveillance system models, embedded system models and requirements models. We have also applied the approach to measuring EMOF metamodels [33]. All these domains are modeled with metamodels that are mutually independent. This shows the independence of the MDM approach with respect to the domain. It also shows its independence w.r.t. the life cycle of systems: from requirements to a final implementation in Java.

To a certain extent, units are domain-dependent (e.g. an integer attribute of a domain class can be a length in meter or a weight in kilogram), therefore they do not appear in the generic metric metamodel. It is left to metric designers to specify core and derived metrics and manipulate numbers in a consistent manner according to their domain expertise. However, note that due to the nature of modeling, all but derived metrics have the most generic scale: an absolute scale [14]. Hence, as noted by [21], the domain expert is allowed to safely use all usual descriptive statistics with the collected metric values.

[5] `http://metrics.sourceforge.net`

	MDM	Mens [31]	Garcìa [15]	Guerra [18]
SigmaMetric	x	x	x	x
PathLengthMetric	x	x	x	x
TauMetric	x	x		x
ValuePerX	x			
SetOfEdgesPerX	x			
SetOfElementsPerX	x			
AboutMeMetric	x			x

Table 4: Supported Types of Metrics

5.7 How expressive is the metric metamodel?

We were able to apply the MDM approach on four different domains (see section 4). In all, we were able to specify 100+ different metrics with the metric metamodel of the MDM approach, as shown table 3. The metric metamodel should be expressive enough for most of the metrics a domain expert can think of.

Table 4 compares the metric types we have identified with those of related work. While a counting metric type like `SigmaMetric` can be found in a lot of previous works, an advanced metric type like `SetOfEdgesPerX` is unique. To sum up, the important contributions of our approach w.r.t. metric types are: 1) metrics are composable together (cf. for instance `DerivedMetric` or `AboutMeMetric` in section 3); 2) the MDM approach has a comprehensive predicate metamodel which supports the expression of complex metrics. None of the previous works details such a predicate metamodel as we have done in this paper.

We have also shown that certain particular metrics can not be expressed with the MDM metamodel (cf. 4.1):

1. if the metric involves domain concepts that are not present in the domain metamodel (e.g. the concept of *pair of functions* in 4.1);

2. if the metric type is too specific to be reused across several independent domains; For instance, let us consider the type-rank metric for object-oriented programs. This metric is the application of the page-rank algorithm of *Google*, to the coupling graph of classes. The commercial tool NDepend[6] implements this metric. We chose deliberately not to include this metric type in the metric metamodel because it is not likely to be understood and reused by domain experts;

3. if the metric involves human activities (e.g. usability metrics);

4. if the metric involves time-dependent properties that can not be expressed in a trace structured by a metamodel (see such traces in 4.4, and §2 in 5.1).

Section 6 will conclude the evaluation of the MDM approach with a detailed comparison against related work.

6 Related Work

Links with the GQM Compared to the Goal/Question/Metric (GQM) approach [3], ours is temporarily after in a measurement plan. To a certain extent, the MDM approach can be considered as an automated implementation of metrics obtained by the GQM approach.

[6] `http://www.ndepend.com/`

Metrics on Top of Metamodels Misic et al. [32] define 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. Along the same line, [45] extends the UML metamodel to provide a basis for metrics expression and then use this model to express known metrics suites with set theory and first order logic. Tang and Chen address [52] the issue of model metrics early in the life cycle. Hence, they present a method to compute object-oriented design metrics from UML models. UML models from Rose are parsed in order to feed their own tool in which metrics are hard-coded. Similarly, El-Wakil et al. [12] use *XQuery* to express metrics for UML class diagrams. Metrics are then computed on XMI files of UML models. On the implementation issue, [20] exposes a concrete design to compute semantic 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.

All these approaches differ from ours on crucial points. They are limited to models of object-oriented software. Hence, they do not address the growing amount of domain-specific languages. Metrics are manually implemented in general-purpose programming languages. This kind of implementation can be considered complex and costly.

Baroni et al. [1] propose to use OCL to express metrics. They use their own metamodel exposed in a previous paper. Likewise, in [30], the authors use Java bytecode to instantiate an object-oriented metamodel and express known cohesion metrics in OCL. OCL, although initially designed to write constraints on UML class diagrams, can be easily extended to define metrics on any domain-metamodel. Also, Reynoso et al. report advances in this direction [46]. OCL and the MDM approach are not disjoint, for instance, a part of the predicate package is similar to some OCL constructs. However, from a conceptual viewpoint, our metric metamodel define the right concepts for specifying model metrics. On the contrary, specifying model metrics with OCL involves the adaption of concepts from OCL to metrics. From a code reduction viewpoint [28] the MDM approach enables us to write shorter metric specifications. The implementation of the C&K metrics in OCL presented in [29] are much longer to the one made here with the MDM approach. However, it is worth studying the mix of the MDM approach and the widely known OCL syntax and constructs to express the predicates.

Finally, Mora et al. [38] shortly describe an original approach to measure models using the Query-View-Transformation language (QVT) (see also [53], where they use the ATL transformation language, which resembles QVT). While both our approach and theses solutions require that domain experts learn a new language, the comprehensiveness of our evaluation significantly differs from [53, 38] and let us validate the properties of our approach (e.g. genericity).

Abstraction Levels for Expressing Metrics Mens et al. define [31] a generic object-oriented metamodel and generic metrics. They then show that known software metrics are an application of these generic metrics. Marinescu et al. propose [28] a simplified implementation of object-oriented design metrics with a metric language called SAIL.

These contributions both explore the possibility of an abstraction level for expressing metrics. However, the domain of application of the abstraction level is limited to object-oriented software metrics. On the contrary, the MDM approach is domain-independent. It can be applied to any imperative or declarative modeling language specified by a metamodel (implementation metamodel, real-time system metamodel, software architecture metamodel, requirements metamodel, system of systems metamodel, etc.).

There are also works about a measurement-related metamodel [16, 15, 42, 39, 17, 50]. A measurement-related metamodel can address three different concerns: the measurement process, the measurement results, and the metric specifications. We address the metamodeling of metric specifications. On the contrary, [16, 15, 42, 39, 17, 50] are more oriented towards metamodeling of measurement processes and measurement results. For instance, important classes of the metamodel of [16] are *ScaleType* and *MeasurementResult*. Also, the unique submission to the OMG request for proposals for a Software

Metrics Meta-Model [42] [7] is very concise on the metamodeling of metric themselves.

To sum up, compared to related work: our metric metamodel is not dedicated to software metrics but to any artifacts specified by a metamodel, it contains more types of metrics (cf. table 4) and; the MDM approach is fully thought in a generative way. The MDM approach remains completely pluggable into measurement frameworks that take different viewpoints (measurement processes and measurement results, interpretation of metric values). We refer to [39, 17, 50] for the latest advances on such frameworks.

Domain Specific metrics The issue of domain specific metrics is a new research field. To our knowledge, it has only been explored by Guerra et al. in [17]. They propose to visually specify metrics for any DSL and introduce a taxonomy of generic metrics. Our approach is similar. We also create a metric specification in order to measure any domain models thanks to code generation.

Since their main goal, visual specification and redesign, is different, we explore different aspects of the issue and study different application cases. We consider metamodels in the EMOF context [41], whereas their metamodels are Entity-Relationships based. We propose new concepts to the metric specification metamodel which enable us more declarative metric specifications.

7 Conclusion

Our model-driven measurement (MDM) approach allows modelers to automatically add measurement capabilities to a domain-specific modeling language. Whatever the domain, whatever the maturity of the product during the development life cycle, it allows the effective measurement thanks to a complete generation of the measurement software from an abstract and declarative specification of metrics. Our prototype is able to generate an integrated and full-fledge domain-specific measurement software from this specification. The MDM approach has been validated through the successful specification of more than 100 metrics in very different domains: software source code, software architecture, systems engineering and requirements.

Future research will continue to question the domain independence of the MDM approach. We are currently studying the measurement of models as different as software processes specified by a software process metamodel; models representing the execution context and environment of software systems (a.k.a. models at runtime); and, in a reflective manner, models representing model transformations.

References

[1] Baroni, A., Braz, S., Abreu, F.: Using OCL to formalize object-oriented design metrics definitions. In: ECOOP'02 Workshop on Quantitative Approaches in OO Software Engineering (2002)

[2] Basili, V.R., Briand, L.C., Melo, W.L.: A validation of object-oriented design metrics as quality indicators. IEEE Trans. Softw. Eng. **22**(10), 751–761 (1996)

[3] Basili, V.R., Caldiera, G., Rombach, H.D.: The goal question metric approach. In: Encyclopedia of Software Engineering. Wiley (1994)

[4] Baumert, J., McWhinney, M.: Software measures and the capability maturity model. Tech. rep., Software Engineering Institute, Carnegie Mellon University (1992)

[7] submitted by Electronic Data Systems, KDM Analytics, Software Revolution, Benchmark Consulting, National Institute of Standards and Technology, `http://www.omg.org/docs/admtf/08-02-01.pdf`

[5] Berenbach, B., Borotto, G.: Metrics for model driven requirements development. In: Proceeding of the 28th International Conference on Software Engineering (ICSE '06), pp. 445–451. ACM Press (2006)

[6] Budinsky, F., Steinberg, D., Merks, E., Ellersick, R., Grose, T.J.: Eclipse Modeling Framework. Addison-Wesley (2004)

[7] Chidamber, S.R., Kemerer, C.F.: Towards a metrics suite for object-oriented design. In: Proceedings of OOPSLA'91, pp. 197–211 (1991)

[8] Coleman, D., Ash, D., Lowther, B., Oman, P.: Using metrics to evaluate software system maintainability. IEEE Computer **27**(8), 44–49 (1994)

[9] Costello, R.J., Liu, D.B.: Metrics for requirements engineering. J. Syst. Softw. **29**(1), 39–63 (1995)

[10] Davis, A., Overmyer, S., Jordan, K., Caruso, J., Dandashi, F., Dinh, A., Kincaid, G., Ledeboer, G., Reynolds, P., Sitaram, P., Ta, A., Theofanos, M.: Identifying and measuring quality in a software requirements specification. In: Proceedings of the First International Software Metrics Symposium (1993)

[11] Douglass, B.P.: Computing model complexity. White paper, I-Logix (2004)

[12] El Wakil, M., El Bastawissi, A., Boshra, M., Fahmy, A.: 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)

[13] Feiler, P.H., Gluch, D.P., Hudak, J.J.: The architecture analysis and design language (aadl): An introduction. Tech. rep., CMU/SEI (2006)

[14] Fenton, N.E.: Software Metrics: A Rigorous Approach. Chapman and Hall (1991)

[15] Garcia, F., Serrano, M., Cruz-Lemus, J., Ruiz, F., Piattini, M.: Managing software process measurement: A metamodel-based approach. Inf. Sci. **177**(12), 2570–2586 (2007)

[16] Garcìa, F., Bertoa, M.F., Calero, C., Vallecillo, A., Ruiz, F., Piattini, M., Genero, M.: Towards a consistent terminology for software measurement. Information & Software Technology **48**(8), 631–644 (2006)

[17] García, F., Ruiz, F., Calero, C., Bertoa, M.F., Vallecillo, A., Mora, B., Piattini, M.: Effective use of ontologies in software measurement. The Knowledge Engineering Review **24**, 23–40 (2009)

[18] Guerra, E., de Lara, J., Díaz, P.: Visual specification of measurements and redesigns for domain specific visual languages. Journal of Visual Languages and Computing **19**, 1–27 (2008)

[19] Hammer, T., Rosenberg, L., Huffman, L., Hyatt, L.: Requirements metrics - value added. In: Proceedings of the 3rd IEEE International Symposium on Requirements Engineering (RE'97), p. 141.1. IEEE Computer Society (1997)

[20] Harmer, T.J., Wilkie, F.G.: An extensible metrics extraction environment for object-oriented programming languages. In: Proceedings of the International Conference on Software Maintenance (2002)

[21] Henderson-Sellers, B.: Object-Oriented Metrics, measures of complexity. Prentice Hall (1996)

[22] Henderson-Sellers, B., Zowghi, D., Klemola, T., Parasuram, S.: Sizing use cases: How to create a standard metrical approach. In: Proceedings of the 8th International Conference on Object-Oriented Information Systems (OOIS '02), pp. 409–421. Springer-Verlag (2002)

[23] Kolde, C.: Basic metrics for requirements management. White paper, Borland (2004)

[24] de Lara, J., Vangheluwe, H.: Atom3: A tool for multi-formalism and meta-modelling. In: Proceedings of the 5th International Conference on Fundamental Approaches to Software Engineering (FASE'02), pp. 174–188. Springer-Verlag (2002)

[25] Lédeczi, A., Bakay, A., Maroti, M., Vőlgyesi, P., Nordstrom, G., Sprinkle, J., Karsai, G.: Composing domain-specific design environments. IEEE Computer **34**, 44–51 (2001)

[26] Loconsole, A.: Measuring the requirements management key process area. In: Proceedings of the 12th European Software Control and Metrics Conference (ESCOM'2001) (2001)

[27] Marchesi, M.: OOA metrics for the Unified Modeling Language. In: Proceedings of the 2nd Euromicro Conference on Software Maintenance and Reengineering (CSMR'98), p. 67. IEEE Computer Society (1998)

[28] Marinescu, C., Marinescu, R., Gîrba, T.: Towards a simplified implementation of object-oriented design metrics. In: IEEE METRICS, p. 11 (2005)

[29] McQuillan, J., Power, J.: A definition of the chidamber and kemerer metrics suite for uml. Tech. rep., National University of Ireland (2006)

[30] McQuillan, J.A., Power, J.F.: 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)

[31] Mens, T., Lanza, M.: A graph-based metamodel for object-oriented software metrics. Electronic Notes in Theoretical Computer Science **72**, 57–68 (2002)

[32] Misic, V.B., Moser, S.: 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)

[33] Monperrus, M.: La mesure des modèles par les modèles : une approche générative. Ph.D. thesis, Université de Rennes (2008)

[34] Monperrus, M., Champeau, J., Hoeltzener, B.: Counts count. In: Proceedings of the 2nd Workshop on Model Size Metrics (MSM'07) co-located with MoDELS'2007 (2007)

[35] Monperrus, M., Jaozafy, F., Marchalot, G., Champeau, J., Hoeltzener, B., Jézéquel, J.M.: Model-driven simulation of a maritime surveillance system. In: Proceedings of the 4th European Conference on Model Driven Architecture Foundations and Applications (ECMDA'2008) (2008)

[36] Monperrus, M., Jézéquel, J.M., Champeau, J., Hoeltzener, B.: Measuring models. In: J. Rech, C. Bunse (eds.) Model-Driven Software Development: Integrating Quality Assurance. IDEA Group (2008)

[37] Monperrus, M., Long, B., Champeau, J., Hoeltzener, B., Marchalot, G., Jézéquel, J.M.: Model-driven architecture of a maritime surveillance system simulator. Systems Engineering **13** (2010)

[38] Mora, B., Garcia, F., Ruiz, F., Piattini, M., Boronat, A., Gómez, A., Carsí, J.Á., Ramos, I.: Software measurement by using qvt transformations in an mda context. In: Proceedings of the International Conference on Enterprise Information Systems (ICEIS'2008) (2008)

[39] Mora, B., Piattini, M., Ruiz, F., Garcia, F.: Smml: Software measurement modeling language. In: Proceedings of the 8th Workshop on Domain-Specific Modeling (DSM'2008) (2008)

[40] Nagappan, N., Ball, T., Zeller, A.: Mining metrics to predict component failures. In: Proceedings of the 28th International Conference on Software Engineering, pp. 452–461. ACM New York, NY, USA (2006)

[41] OMG: MOF 2.0 specification. Tech. rep., Object Management Group (2004)

[42] OMG: Rfp for a software metrics meta-model (adtmf/2006-09-03). Tech. rep., OMG (2006)

[43] Paulk, M.C., Weber, C.V., Garcia, S.M., Chrissis, M.B., Bush, M.: Key practices of the capability maturity model. Tech. rep., Software Engineering Institute (1993)

[44] Pawlak, R., Noguera, C., Petitprez, N.: Spoon: Program analysis and transformation in java. Tech. Rep. 5901, INRIA (2006)

[45] Reissing, R.: Towards a model for object-oriented design measurement. In: ECOOP'01 Workshop QAOOSE (2001)

[46] Reynoso, L., Genero, M., Cruz-Lemuz, J., Piattini, M.: OCL 2: Using OCL in the Formal Definition of OCL Expression Measures. In: Proceedings of the 1st Workshop on Quality in Modeling (QIM'2006) (2006)

[47] SAE: AADL Standard. Tech. rep., Society of Automotive Engineers (2006)

[48] Schmidt, D.C.: Model-driven engineering. IEEE Computer **39**(2), 25–31 (2006)

[49] Singh, Y., Sabharwal, S., Sood, M.: A systematic approach to measure the problem complexity of software requirement specifications of an information system. Information and Management Sciences **15**, 69–90 (2004)

[50] Staron, M., Meding, W., Nilsson, C.: A framework for developing measurement systems and its industrial evaluation. Information and Software Technology **51**, 721–737 (2009)

[51] Sztipanovits, J.: Advances in model-integrated computing. In: Proceedings of the 18th IEEE Instrumentation and Measurement Technology Conference (IMTC'2001), vol. 3, pp. 1660–1664 (2001)

[52] Tang, M.H., Chen, M.H.: Measuring OO design metrics from UML. In: Proceedings of MODELS/UML'2002. UML 2002 (2002)

[53] Vépa, E., Bézivin, J., Brunelière, H., Jouault, F.: Measuring model repositories. In: Proceedings of the 1st Workshop on Model Size Metrics (MSM'06) co-located with MoDELS'2006 (2006)

[54] Zave, P.: Classification of research efforts in requirements engineering. ACM Comput. Surv. **29**(4), 315–321 (1997)