



A Model-driven Measurement Approach

Martin Monperrus^{1,2}, Jean-Marc Jézéquel², Joël Champeau¹, and Brigitte Hoeltzener¹

1. ENSIETA - Brest (Fr) . 2. INRIA & University of Rennes (Fr)

Abstract. Companies using domain specific languages in a model-driven development process need to measure their models. However, developing and maintaining a measurement software for each domain specific modeling language is costly. Our contribution is a model-driven measurement approach. This measurement approach is model-driven from two viewpoints: 1) it measures models of a model-driven development process; 2) it uses models as unique and consistent metric specifications, w.r.t a metric specification metamodel. This declarative specification of metrics is then used to generate a fully fledged implementation. The benefit from applying the approach is evaluated by two applications. They indicate that this approach reduces the domain-specific measurement software development cost.

1 Introduction

Model-driven engineering (MDE) [1] is an approach to software development that advocates the creation of domain-specific languages (DSL). Contrary to general-purpose programming languages, a domain-specific language addresses a limited range of applications. The main goal of DSLs is to reduce the product development cost by means of generative techniques [2]. From a MDE viewpoint, the DSL is specified with a metamodel; a program written in a DSL is called a model.

To address safety-critical concerns or quality assurance, models need to be measured. Ledeczki et al. state [2], that *another 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. [1]: *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 a big enough niche market; that is to say there are no measurement software vendors for specific domains. Hence, companies have to fully support the development cost of the model measurement software.

Conversely, generative techniques (e.g. [3]) are used to generate most of the modeling environment from an abstract definition of the domain, called a meta-model. These techniques allow a low-cost modeling environment. Systems are produced by means of model-driven code generation, as well as modeling environment. In this context, it is not conceivable to manually develop the measurement software. Indeed, the NASA recommends that *the cost of measurement should not add more than 2 percent to the software development or maintenance effort* [4].

Classical object-oriented program measurement software packages are complex and costly [5]. Similarly, the cost of a DSL measurement software comes from several requirements which are mainly: the automation of measurement, the integration into a modeling tool, the communication between the domain expert and developers, the need for extensibility and tailoring.

Our goal is to address the cost of a DSL measurement software by providing a generative measurement approach that can be applied to any kind of models. In other words, we would like to be able to generate a measurement software for design models, architectural models, performance models, requirements model, etc.

Our contribution is a model-driven measurement approach (we will use the expression *MDM approach* later in the paper). The core of the contribution is a metric specification metamodel. This measurement approach is model-driven from two viewpoints: 1) it measures models of a model-driven development process; 2) it uses models as unique and consistent metric specifications, w.r.t the metric specification metamodel. This approach consists of specifying metrics as instance of the metric specification metamodel. These metric specifications refer to a domain metamodel. The metric specifications are used to generate a fully fledged measurement software. The generated artifacts fully fulfill the requirements for measurement software presented above. The benefit from applying the MDM approach is evaluated with two applications.

The remainder of this paper is organized as follows. In section 2, we present the full process of model-driven measurement. Then, we present the metric specification metamodel (section 3). The instantiation of the whole approach is presented in two different contexts (section 4), including in an industrial one. We finally discuss related works in section 5 and conclude.

2 Overview of the MDM Approach

The measurement approach presented in this paper is intended to be applied to models of a MDE software development i.e.; applied to models fully described by a metamodel, which is later called domain metamodel. Our intuition was [6] that it is possible to generate a 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.

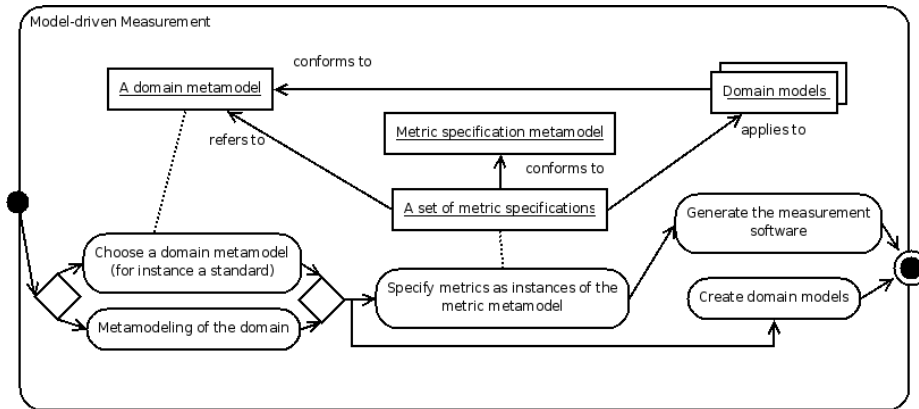


Fig. 1. Model-driven measurement: actions and artifacts

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 into 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.; 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. Note that the same domain models can also ground other engineering activities such as code generation, verification and validation, or model transformations.

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 [7, 8, 3]. 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 fully fledged 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 (a generic development environment, see www.eclipse.org) and EMF [3] 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 a 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 architecture models or requirements models. In that sense, our metric specification metamodel is domain-independent [9]. It has been built following a bottom-up approach: from existing metrics to concepts of the metamodel. In regard to the application cases we studied, no other parts are needed.

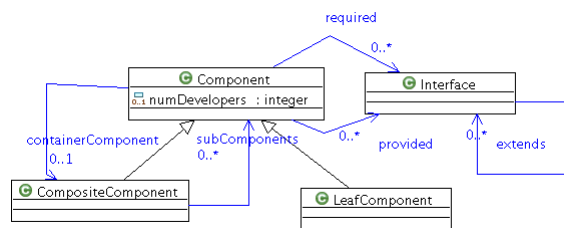


Fig. 2. The domain metamodel used for demonstration purposes

Supporting example and syntax The presentation of the metric specification metamodel will be helped by several examples. Note that the concepts of the metric specification metamodel will be further called classes. Main ones will be emphasized as italic. Examples consists 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.

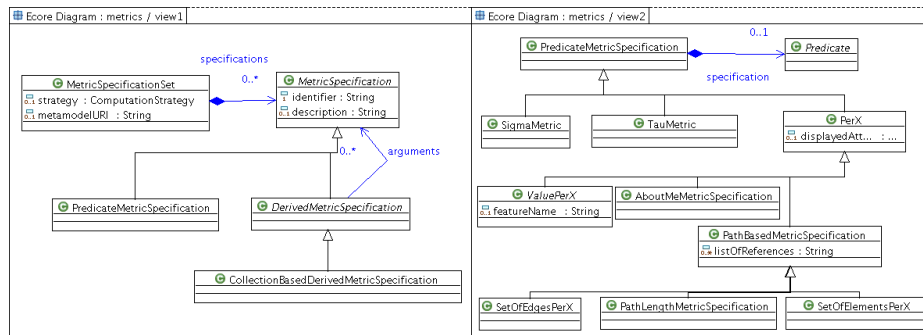


Fig. 3. The core of the metric specification metamodel

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 domain model are measured. This can be a file, a predefined list of model elements or a transitive closure on all model elements. 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 subclassed. The `DerivedMetricSpecification` class handles arithmetic and based function based metrics (e.g.; addition, exponential): the subclasses 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. It is subclassed as `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 comes the declarative part of the metric. For instance, a SigmaMetric declares a predicate and a TauMetric a reference name. Note that the predicate is a textual syntax of predicate-based model elements, that will be presented later.

```

1 metric SigmaMetric NOCompComp is
2   description "too complex components"
3   elements satisfy "this.isInstance(Component)
4     and this.required > 10
5     and this.provided > 10
6     and this.subComponents > 5"
7 endmetric
8
9 metric TauMetric NDeComp is
10  description "The number of decomposition"
11  link is "CompositeComponent:subComponent"
12 endmetric

```

The PerX class is an abstract class to specify metrics that are related to a given model element. By definition, the measurement of a domain model thanks to a PerX metric specification gives several values. 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 subclassed 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.

```

1 metric AttributeValuePerX NDev is
2   description "The number of developers per component"
3   elements satisfy "this.isInstance(Component)"
4   value is "this.numDevelopers"
5 endmetric
6
7 metric AverageMetric ANORI is
8   description "The average number of required
9     interfaces per component"
10  input metric MultiplicityPerX is
11  elements satisfy "this.isInstance(Component)"
12  multiplicity of "this.required"
13 endmetric
14 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.

An AboutMeMetricSpecification is a metric that considers a given object as primary variable. This metric type is used each time the domain model contains derived relationships. That is to say, when a metric is the number of elements that has something to do with an object (symbolized by *me* in the name), but when there is not the corresponding reference is the domain metamodel.

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.

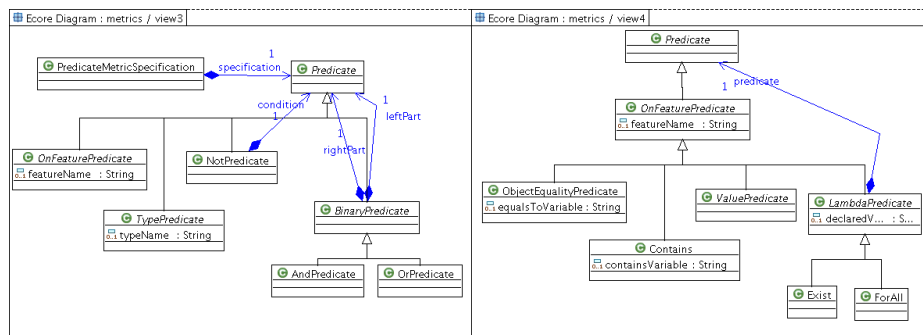


Fig. 4. Predicates in the metric specification metamodel

```

1 metric SetOfElementsPerX NOIT is
2   description "The number of interfaces
3     involved per component"
4   x satisfy "this.isInstance(Component)"
5   elements satisfy "this.isInstance(Interface)"
6   references followed "required,provided,extends"
7 endmetric
8
9 metric AboutMeMetricSpecification COUPLING is
10  description "coupling to this interface"
11  elements satisfy "Interface.isInstance(self)"
12  internal metric spec is metric SigmaMetric COUPLING_ is
13  description "used for COUPLING"
14  elements satisfy "this.provided == __me__ or this.required == __me__ "
15 endmetric
16
17 metric PathLengthMetricSpecification DD is
18  description "Depth in decomposition"
19  x satisfy "this.isInstance(LeafComponent)"
20  references followed "container"
21 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 subpredicates, in the same manner as a boolean function. Therefore, the metamodel contains the `AndPredicate`, `OrPredicate`, and `NotPredicate`. There are also classes to handle tests on the type of the model element. They are subclasses of the `TypePredicate` class. `IsInstance` tests the metaclass of a model element; `IsDirectInstance` tests the metaclass or one of its superclasses. 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 are the possible tests on a given feature. `ObjectEqualityPredicate` enables us to test whether a reference points to a model element referred by a variable. `ValuePredicate` enables 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 subclassed 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 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.

Limitations of the MDM approach Since the metamodel is declarative, all metrics that are defined on top of explicit concepts of the metamodel can generally be specified within the MDM approach. The reciprocal statement is also true: any metrics that is composed of concepts not present in the metamodel cannot be expressed within the MDM approach. This point will be illustrated in the next section.

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 Study of Instances of the MDM Approach

In this section we present the application of the MDM approach into two different contexts i.e.; the instantiation of the metric specification metamodel for two different domain metamodels.

4.1 Considering a Java program as an implementation model

Let us consider Java programs as models conforming to Java metamodel. We can then specify Chidamber and Kemerer metrics [10] within the MDM approach. This viewpoint will help to demonstrate that 1) our metric specification metamodel encompasses with existing software metrics and 2) the code volume of the metrics implemented within the MDM approach is much smaller than the volume of the final executable code.

We have used the SpoonEMF (<http://tinyurl.com/spoonemf>) tool to transform a Java software to an Eclipse/EMF model. SpoonEMF is a binding between Spoon and EMF. Spoon [11] 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 into a single XMI model file.

We will now present the specification of the Chidamber and Kemerer metrics [10] as instance of our metric specification metamodel referring to the Spoon/Java metamodel. It is a fully declarative specification.

Weighted Methods per Class: We consider the Basili et al's [12] version of WMC, which is the number of methods defined in each class.

```

1 metric MultiplicityPerX WMC is
2   description "Weighted Methods per Class"
3   elements satisfy "this.isInstance(CtClass)"
4   reference is "Methods"
5 endmetric

```

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

```

1 metric PathLengthMetricSpecification DIT is
2   description "Depth in Inheritance Tree per Class"
3   x satisfy "this.isInstance(CtClass)"
4   references followed "Superclass"
5 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 not the reference *children* in CtClass. AboutMeMetricSpecification fills this lack and enables us to correctly specify the metric.

```

1 metric AboutMeMetricSpecification NOC is
2   description "Number of Children of a Class"
3   x satisfy "this.isInstance(CtClass)"
4   input metric SigmaMetric _NOC is
5   elements satisfy "this.Superclass == __me__"
6   endmetric
7 endmetric

```

Coupling Between Object Classes: CBO is the number of classes to which a given class is coupled.

```

1 metric AboutMeMetricSpecification CBO is
2   description "Coupling Between Object Classes"
3   elements satisfy "this.isInstance(CtClass)"
4   input metric SetOfEdgesPerX _CBO is
5   target satisfy "this.isInstance(CtTypeReference)
6     and this.QualifiedName == __me__.SimpleName"
7   endmetric
8 endmetric

```

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

```

1 metric SetOfElementsPerX RFC is
2   description "Response For a Class"
3   elements satisfy "this.isInstance(CtClass)"
4   x satisfy "this.isInstance(CtExecutableReference)"
5   references followed "Methods,Body,Statements,
6     Expression,AssertExpression,CaseExpression,"

```

```

7     ThenStatement , Condition , ElseStatement , Selector ,
8     Cases , Block , Finalizer , Catchers , AssertExpression ,
9     CaseExpression , Finalizer , Executable ,
10    DeclaringExecutable"
11 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.

The benefits 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.

The MDM approach is generative. The specification of Chidamber and Kemerer metrics 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 sits on top of SpoonEMF/EMF/Eclipse. It is not possible to estimate precisely the size of the actually used part of these software packages. By comparison, the *Metrics* (<http://metrics.sourceforge.net>) Java measurement tool is more than manually coded 20000 lines of code (its maturity and completeness has to be taken into account). Note that Java is a verbose language. With a more compact generated language like Smalltalk, this generated code volume would decrease but without the conciseness of pure dedicated declarative metric concepts.

4.2 Industrial application: metrics for maritime surveillance system models

In this section, we present the application of the MDM approach in an industrial context. Our industrial partner, Thales Airborne Systems, is a world-wide company that designs and develops maritime surveillance systems. The MDM approach applied early in the system life cycle enables engineers to obtain: 1) metric values that estimate operational performance of the system; 2) architectural metric values that are used to dimension the system.

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 functionalities, the relationships between hardware and software components and the communication between the system and others entities (base, other MSSs) indicate the complexity of the system.

Current system engineering process are mainly document-centric. Current MSS simulators are mainly code centric. During the system engineering activities, the problem is that it is very costly to develop a measurement software on top of both documents and code, in order to obtain operational metrics for MSS.

This context of system engineering metrics for MSS is a good candidate to apply the MDM approach. Therefore, we tried to obtain MSS domain metrics with our approach. In a previous paper [13], we presented a model-driven simulator for maritime surveillance systems. In this paper, we measure the models involved in the model-driven simulation with the MDM approach.

There is 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 models can be found in [13]. 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.

Domain 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 In the context of maritime surveillance system development at the system engineering level, we presented the application of the MDM approach. We presented the main components of the approach: the domain metamodels, models and metrics. The MDM approach enables engineers to obtain domain metric values for three different type of models used during the system engineering activities. This comprehensive and consistent way of measuring system

models is an added value compared to costly ad hoc developments which consider either too document-centric or too code-centric artifacts.

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. However, this dramatic cost reduction factor has to be mitigated by the differences in maturity level and reliability of the obtained software. Still, it is clear that the MDM approach reduces the development cost by an order of magnitude (when creating the domain metamodel from scratch or to fit an existing DSL, the cost of its creation has also to be taken into account).

The success of this project is a first validation of the MDM approach. It also shows that measurement is not orthogonal to other engineering activities. The measured models, which give an estimation of the future characteristics of the system being built, are the same that are used for simulation. We assume that they could also be used for other model-driven activities such as model transformations, or model-driven testing.

5 Related Work

Links with the GQM Compared to the Goal/Question/Metric (GQM) approach [14], 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.

Metrics on top of metamodels Misic et al. [15] 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, [16] 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 [17] 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. [18] use XQuery to express metrics for UML class diagrams. Metrics are then computed on XMI files of UML models. On the implementation issue, [19] 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 differs 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 a general-purpose programming languages. This kind of implementation can be considered complex and costly.

Baroni et al. [20] propose to use OCL to express metrics. They use their own metamodel exposed in a previous paper. Likewise, in [21], the authors use Java bytecode to instantiate the Dagstuhl 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.

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 [5] the MDM approach enables to write shorter metric specifications. The implementation of the C&K metrics in OCL made in [22] 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.

Abstraction level for expressing metrics Mens et al. define [23] 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 [5] 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 previous contributions about a measurement-related metamodel. It is to be noted that a measurement-related metamodel can address three different concerns: the measurement process, the measurement results, and the metric specifications. The metamodel of [24] is more about the two first points (e.g.; classes InformationNeed, ScaleType, MeasurementResult). The OMG requested proposals for a Software Metrics Meta-Model ¹. Very recently, the unique submission was published ². This RFP was not very precise on which aspect the metrics metamodel is intended, but the proposed metamodel explicitly supports the three aspects.

Compared to these contributions, the MDM approach has a smaller scope: the metric specifications. Our declarative ways of specifying metrics are novel (except SigmaMetricSpecification and PathLengthMetricSpecification [23]) and are not dedicated to software metrics but to any artifacts specified by a metamodel. The main difference is that the MDM approach is fully thought in a generative way.

¹ OMG document adtmf/2006-09-03

² OMG document adtmf/2008-02-01

Domain specific metrics The issue of domain specific metrics is a very new research field. As of today and to our knowledge, it has only been explored by Guerra et al. very recently in [25]. 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 [26], whereas their metamodels are Entity-Relationships based. We propose new concepts to the metric specification metamodel which enable us more declarative metric specifications.

6 Conclusion

Our Model-driven Measurement Approach is a new technique in the field of measurement and model-driven engineering. 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 metric specification.

This generative approach cuts the development cost of measurement software. From a code volume viewpoint, the ratio between the final implementation and the abstract metric specifications is more than 10. At the system engineering level, it allows measurement not really made before.

On this basis, future research will be conducted to empirically measure the productivity enhancement achieved by a model-driven measurement approach. An important milestone will be to define an intuitive way of specifying metrics conform to the metric specification metamodel so as to let domain experts writing metrics by themselves.

References

1. Schmidt, D.C.: Model-driven engineering. *IEEE Computer* **39**(2) (February 2006) 25–31
2. Lédeczi, A., Bakay, A., Maroti, M., Völgyesi, P., Nordstrom, G., Sprinkle, J., Karsai, G.: Composing domain-specific design environments. *IEEE Computer* **34** (November 2001) 44–51
3. Budinsky, F., Steinberg, D., Merks, E., Ellersick, R., Grose, T.J.: *Eclipse Modeling Framework*. Addison-Wesley (2004)
4. NASA Software Program: *Software measurement guidebook*. Technical report, National Aeronautics and Space Administration (1995)
5. Marinescu, C., Marinescu, R., Gîrba, T.: Towards a simplified implementation of object-oriented design metrics. In: *IEEE METRICS*. (2005) 11
6. Monperrus, M., Champeau, J., Hoeltzner, B.: Counts count. In: *Proceedings of the 2nd Workshop on Model Size Metrics (MSM'07) co-located with MoDELS'2007*. (2007)

7. Sztipanovits, J.: Advances in model-integrated computing. In: Proceedings of the 18th IEEE Instrumentation and Measurement Technology Conference (IMTC'2001). Volume 3. (2001) 1660–1664
8. 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), Springer-Verlag (2002) 174–188
9. Monperrus, M., Jézéquel, J.M., Champeau, J., Hoeltzner, B.: Measuring models. In Rech, J., Bunse, C., eds.: Model-Driven Software Development: Integrating Quality Assurance. IDEA Group (2008)
10. Chidamber, S.R., Kemerer, C.F.: Towards a metrics suite for object-oriented design. In: Proceedings of OOPSLA'91. (1991) 197–211
11. Pawlak, R., Noguera, C., Petitprez, N.: Spoon: Program analysis and transformation in java. Technical Report 5901, INRIA (2006)
12. 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) (1996) 751–761
13. Monperrus, M., Jaozafy, F., Marchalot, G., Champeau, J., Hoeltzner, 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)
14. Basili, V.R., Caldiera, G., Rombach, H.D.: The goal question metric approach. In: *Encyclopedia of Software Engineering*. Wiley (1994)
15. Misis, V.B., Moser, S.: From formal metamodells to metrics: An object-oriented approach. In: Proceedings of the Technology of Object-Oriented Languages and Systems Conference (TOOLS'97). (1997) 330
16. Reissing, R.: Towards a model for object-oriented design measurement. In: ECOOP'01 Workshop QAOOSE. (2001)
17. Tang, M.H., Chen, M.H.: Measuring OO design metrics from UML. In: Proceedings of MODELS/UML'2002, UML 2002 (2002)
18. 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)
19. 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)
20. 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)
21. 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)
22. McQuillan, J., Power, J.: A definition of the chidamber and kemerer metrics suite for uml. Technical report, National University of Ireland (2006)
23. Mens, T., Lanza, M.: A graph-based metamodel for object-oriented software metrics. *Electronic Notes in Theoretical Computer Science* **72** (2002) 57–68
24. García, 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) (2006) 631–644
25. Guerra, E., de Lara, J., Díaz, P.: Visual specification of measurements and redesigns for domain specific visual languages. *Journal of Visual Languages and Computing in Press* (nov 2007) 1–27
26. OMG: MOF 2.0 specification. Technical report, Object Management Group (2004)